# Concurrency Theory

**Winter Semester 2015/16**

**Lecture 10: Variations of $\pi$-Calculus**

**Joost-Pieter Katoen and Thomas Noll**
**Software Modeling and Verification Group**
**RWTH Aachen University**

http://moves.rwth-aachen.de/teaching/ws-1516/ct/

## Syntax of the Monadic $\pi$-Calculus

### Definition (Syntax of monadic $\pi$-Calculus)

- Let $A = \{a, b, c \ldots, x, y, z, \ldots\}$ be a set of names.
- The set of action prefixes is given by

$$
\begin{aligned}
\pi ::=\ & x(y) && \text{(receive } y \text{ along } x\text{)} \\
\mid\ & \overline{x}\langle y \rangle && \text{(send } y \text{ along } x\text{)} \\
\mid\ & \tau && \text{(unobservable action)}
\end{aligned}
$$

- The set $Prc^{\pi}$ of $\pi$-Calculus process expressions is defined by the following syntax:

$$
\begin{aligned}
P ::=\ & \textstyle\sum_{i \in I} \pi_i.P_i && \text{(guarded sum)} \\
\mid\ & P_1 \parallel P_2 && \text{(parallel composition)} \\
\mid\ & \text{new}\, x\, P && \text{(restriction)} \\
\mid\ & !P && \text{(replication)}
\end{aligned}
$$

(where $I$ finite index set, $x \in A$)

**Conventions:** $\text{nil} := \sum_{i \in \emptyset} \pi_i.P_i$, $\text{new}\, x_1, \ldots, x_n\, P := \text{new}\, x_1\, (\ldots \text{new}\, x_n\, P)$

## A Standard Form

**Theorem (Standard form)**

*Every process expression is structurally congruent to a process of the standard form*

$$\text{new } x_1, \ldots, x_k \, (P_1 \parallel \ldots \parallel P_m \parallel !Q_1 \parallel \ldots \parallel !Q_n)$$

*where each $P_i$ is a non-empty sum, and each $Q_j$ is in standard form.*

*(If $m = n = 0$: nil; if $k = 0$: restriction absent)*

**Proof.**

by induction on the structure of $R \in Prc^\pi$ (on the board) □

## The Reaction Relation

Thanks to Theorem 9.5, only processes in standard form need to be considered for defining the operational semantics:

### Definition

The reaction relation $\longrightarrow \subseteq Prc^\pi \times Prc^\pi$ is generated by the rules:

$$\text{(Tau)}\frac{}{\tau.P + Q \longrightarrow P}$$

$$\text{(React)}\frac{}{(x(y).P + R) \parallel (\overline{x}\langle z\rangle.Q + S) \longrightarrow P[z/y] \parallel Q}$$

$$\text{(Par)}\frac{P \longrightarrow P'}{P \parallel Q \longrightarrow P' \parallel Q} \qquad \text{(Res)}\frac{P \rightarrow P'}{\text{new } x\, P \longrightarrow \text{new } x\, P'}$$

$$\text{(Struct)}\frac{P \longrightarrow P'}{Q \longrightarrow Q'} \quad \text{if } P \equiv Q \text{ and } P' \equiv Q'$$

- $P[z/y]$ replaces every free occurrence of $y$ in $P$ by $z$.
- In (React), the pair $(x(y), \overline{x}\langle z\rangle)$ is called a redex.



Software Modeling and Verification Chair RWTH AACHEN UNIVERSITY

## Example: Mobile Clients

### Example 10.1

- System specification (cf. Example 8.10):

$$
\begin{aligned}
System_1 &= \text{new}\, L\,(Client_1 \parallel Station_1 \parallel Idle_2 \parallel Control_1)\\
System_2 &= \text{new}\, L\,(Client_2 \parallel Idle_1 \parallel Station_2 \parallel Control_2)\\
Station(talk, switch, gain, lose) &= talk.Station(talk, switch, gain, lose) +\\
&\quad lose(t, s).\overline{switch}\langle t, s\rangle.Idle(gain, lose)\\
Idle(gain, lose) &= gain(t, s).Station(t, s, gain, lose)\\
Control_1 &= \overline{lose_1}\langle talk_2, switch_2\rangle.\overline{gain_2}\langle talk_2, switch_2\rangle.Control_2\\
Control_2 &= \overline{lose_2}\langle talk_1, switch_1\rangle.\overline{gain_1}\langle talk_1, switch_1\rangle.Control_1\\
Client(talk, switch) &= \overline{talk}.Client(talk, switch) + switch(t, s).Client(t, s)\\
L &= (talk_i, switch_i, gain_i, lose_i \mid i \in \{1, 2\})
\end{aligned}
$$

- Use additional reaction rule for polyadic communication:

$$
\text{(React')}\;\frac{}{(x(\vec{y}).P + R) \parallel (\overline{x}\langle\vec{z}\rangle.Q + S) \longrightarrow P[\vec{z}/\vec{y}] \parallel Q}
$$

- Use additional congruence rule for process calls: if $A(\vec{x}) = P_A$, then $A(\vec{y}) \equiv P_A[\vec{y}/\vec{x}]$
- Show $System_1 \longrightarrow^* System_2$ (on the board)

## Polyadic Communication I

- **So far:** messages with exactly one name
- **Now:** arbitrary number
- New types of action prefixes:

$$x(y_1, \ldots, y_n) \qquad \text{and} \qquad \overline{x}\langle z_1, \ldots, z_n\rangle$$

  where $n \in \mathbb{N}$ and all $y_i$ distinct
- Expected behavior:

$$\text{(React')}\frac{}{(x(\vec{y}).P + R) \parallel (\overline{x}\langle\vec{z}\rangle.Q + S) \longrightarrow P[\vec{z}/\vec{y}] \parallel Q}$$

  (replacement of free names)
- Obvious attempt for encoding:

$$x(y_1, \ldots, y_n).P \mapsto x(y_1) \ldots x(y_n).P$$
$$\overline{x}\langle z_1, \ldots, z_n\rangle.Q \mapsto \overline{x}\langle z_1\rangle \ldots \overline{x}\langle z_n\rangle.Q$$

## Polyadic Communication II

- But consider the following counterexample.

  Polyadic representation:

$$x(y_1, y_2).P \parallel \overline{x}\langle z_1, z_2\rangle.Q \parallel \overline{x}\langle z_1', z_2'\rangle.Q'$$

$$\swarrow \quad \searrow$$

$$P[z_1/y_1, z_2/y_2] \parallel Q \parallel \overline{x}\langle z_1', z_2'\rangle.Q' \qquad P[z_1'/y_1, z_2'/y_2] \parallel \overline{x}\langle z_1, z_2\rangle.Q \parallel Q'$$

  Monadic encoding: $P[z_1/y_1, z_2/y_2] \parallel \ldots \quad \checkmark \qquad P[z_1'/y_1, z_2'/y_2] \parallel \ldots \quad \checkmark$

$$\uparrow^2 \qquad\qquad \uparrow^2$$

$$x(y_1).x(y_2).P \parallel \overline{x}\langle z_1\rangle.\overline{x}\langle z_2\rangle.Q \parallel \overline{x}\langle z_1'\rangle.\overline{x}\langle z_2'\rangle.Q'$$

$$\downarrow_2 \qquad\qquad \downarrow_2$$

$$P[z_1/y_1, z_1'/y_2] \parallel \ldots \quad ↯ \qquad P[z_1'/y_1, z_1/y_2] \parallel \ldots \quad ↯$$

- **Solution:** avoid interferences by first introducing a fresh channel:

$$x(y_1, \ldots, y_n).P \mapsto x(w).w(y_1)\ldots w(y_n).P$$
$$\overline{x}\langle z_1, \ldots, z_n\rangle.Q \mapsto \text{new } w\,(\overline{x}\langle w\rangle.\overline{w}\langle z_1\rangle\ldots\overline{w}\langle z_n\rangle.Q)$$

  where $w \notin fn(Q) \cup \{y_1, \ldots, y_n, z_1, \ldots, z_n\}$

- **Correctness:** see exercises

## Adding Recursive Process Calls

### Recursive Process Calls I

- **So far:** process replication $!P$
- **Now:** parametric process definitions of the form

$$A(x_1, \dots, x_n) = P_A$$

  where $A$ is a process identifier and $P_A$ a process expression containing calls of $A$ (and possibly other parametric processes)
- Semantic interpretation by new congruence rule:

$$A(y_1, \dots, y_n) \equiv P_A[y_1/x_1, \dots, y_n/x_n]$$

- Again: possible to simulate in basic calculus by using
  - message passing to model parameter passing to $A$
  - replication to model the multiple activations of $A$
  - restriction to model the scope of the definition of $A$

## Recursive Process Calls II

The encoding

- of a process definition $A(\vec{x}) = P_A$
- with right-hand side $P_A = \ldots A(\vec{u}) \ldots A(\vec{v}) \ldots$
- for main process $Q = \ldots A(\vec{y}) \ldots A(\vec{z}) \ldots$

is defined as follows:

1. Let $a \in A$ be a new name (standing for $A$).
2. For any process $R$, let $\hat{R}$ be the result of replacing every call $A(\vec{w})$ by $\overline{a}\langle\vec{w}\rangle.\mathrm{nil}$.
3. Replace $Q$ by $Q' := \mathrm{new}\, a\,(\hat{Q} \parallel\, !a(\vec{x}).\hat{P}_A)$.

(In the presence of more than one process identifier, $Q'$ will contain a replicated component for each definition.)

**Example 10.2**

One-place buffer:

$$B(in, out) = in(x).\overline{out}\langle x\rangle.B(in, out)$$

(on the board)

## Asynchronous Communication

- So far: CCS and $\pi$-Calculus feature synchronous communication: interaction involves joint participation of processes ("handshaking")
- Prefix operator expresses temporal precedence:
  - $\overline{x}\langle y\rangle.P$ requires $y$ to be received before executing $P$
  - $x(z).Q$ requires (of course) $z$ to be sent before executing $Q$
- But: many concurrent (especially distributed) systems use asynchronous communication where sending and receiving are separated: sender can continue before actual reception
- Often involves explicit medium of certain characteristic
  - bounded or unbounded capacity
  - preserving sending order or not
- Now: introduce subcalculus of $\pi$-Calculus with asynchronous communication
- "Trick": output prefix can only be followed by nil
  - (unguarded) subprocess $\overline{x}\langle y\rangle.\mathrm{nil}$ ("output particle") can be understood as message $y$ in (implicit) communication medium
  - available for reception to any (unguarded) subprocess of the form $x(z).Q$
  - $y$ is sent when $\overline{x}\langle y\rangle.\mathrm{nil}$ becomes unguarded
  - $y$ is received when $\overline{x}\langle y\rangle.\mathrm{nil}$ disappears via reaction $\overline{x}\langle y\rangle.\mathrm{nil} \parallel (x(z).Q + R) \longrightarrow Q[y/z]$

  $\Longrightarrow$ syntactic modification sufficient, no change of semantics

# The Asynchronous $\pi$-Calculus I

## Definition 10.3 (Syntax of asynchronous $\pi$-Calculus)

- Let $A = \{a, b, c \ldots, x, y, z, \ldots\}$ be a set of names.
- The set of action prefixes is given by

$$\begin{array}{rll} \pi ::= & x(y) & \text{(receive } y \text{ along } x\text{)} \\ \mid & \tau & \text{(unobservable action)} \end{array}$$

- The set $Prc^{a\pi}$ of asynchronous $\pi$-Calculus process expressions is defined by the following syntax:

$$\begin{array}{rll} P ::= & \sum_{i \in I} \pi_i.P_i & \text{(guarded sum)} \\ \mid & \overline{x}\langle y \rangle.\text{nil} & \text{(asynchronous output)} \\ \mid & P_1 \parallel P_2 & \text{(parallel composition)} \\ \mid & \text{new}\, x\, P & \text{(restriction)} \\ \mid & !P & \text{(replication)} \end{array}$$

(where $I$ finite index set, $x, y \in A$)

## The Asynchronous $\pi$-Calculus II

- As $Prc^{a\pi} \subseteq Prc^{\pi}$, the semantics of the asynchronous $\pi$-Calculus does not have to be defined explicitly
- $Prc^{a\pi}$ actually imposes two restrictions:
  - output particles can only be followed by nil (as discussed before)
  - output particles cannot be summands in an expression $\sum_{i \in I} \pi_i.P_i$ where $|I| > 1$
- Second restriction also in line with asynchronous communication:
  - (unguarded) particle $\overline{x}\langle y\rangle.\mathrm{nil}$ represents message that *has been* sent
  - process like $\overline{x}\langle y\rangle.\mathrm{nil} + v(w).Q$ is *capable* of sending via $x$, but also capable of receiving via $v$ (which disables sending)
  - thus: correspondence between sent (but unreceived) message and presence of (unguarded) output particle would get lost

## Encoding Synchronous Communication

- Synchronous communication: sender only allowed to continue if message has been received
- Usual asynchronous implementation: enforce synchronous behaviour by two asynchronous communication operations
  - sending of actual data
  - waiting for acknowledgement
- Here: encoding carried out in two steps
  1. encoding (monadic) synchronous by polyadic asynchronous communication
  2. encoding polyadic asynchronous by monadic asynchronous communication

## Encoding Synchronous by Polyadic Asynchronous Communication

- **Encoding:**
  - sending: $\overline{x}\langle y\rangle.P \mapsto \text{new } v\,(\overline{x}\langle y, v\rangle.\text{nil} \parallel v().P)$
  - receiving: $x(z).Q \mapsto x(z, v).(\overline{v}\langle\rangle.\text{nil} \parallel Q)$

  where $v \notin fn(P) \cup fn(Q) \cup \{x, y\}$ ("acknowledgement channel")
- **Correctness:** synchronous transition

  $$\overline{x}\langle y\rangle.P \parallel x(z).Q \longrightarrow P \parallel Q[y/z]$$

  is mimicked by polyadic asynchronous transition sequence

  $$\begin{array}{rll} & \text{new } v\,(\overline{x}\langle y, v\rangle.\text{nil} \parallel v().P) \parallel x(z, v).(\overline{v}\langle\rangle.\text{nil} \parallel Q) & \text{(encoding)} \\ \equiv & \text{new } v\,(\overline{x}\langle y, v\rangle.\text{nil} \parallel v().P \parallel x(z, v).(\overline{v}\langle\rangle.\text{nil} \parallel Q)) & \text{(scope extension)} \\ \longrightarrow & \text{new } v\,(v().P \parallel \overline{v}\langle\rangle.\text{nil} \parallel Q[y/z]) & \text{(reaction)} \\ \longrightarrow & \text{new } v\,(P \parallel Q[y/z]) & \text{(reaction)} \\ \equiv & P \parallel Q[y/z] & \text{(congruence)} \end{array}$$

- **Note:** $P$ only executable after completion of $Q$'s input



Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

## Encoding Polyadic by Monadic Asynchronous Communication

- **Encoding:** (for two parameters, using $v/w$ for sender from/to receiver)
  - sending: $\overline{x}\langle y_1, y_2\rangle.\text{nil} \mapsto \text{new } v\,(\overline{x}\langle v\rangle.\text{nil} \parallel v(w).(\overline{w}\langle y_1\rangle.\text{nil} \parallel v(w).\overline{w}\langle y_2\rangle.\text{nil}))$
  - receiving: $x(z_1, z_2).R \mapsto x(v).\text{new } w\,(\overline{v}\langle w\rangle.\text{nil} \parallel w(z_1).(\overline{v}\langle w\rangle.\text{nil} \parallel w(z_2).R))$

  where $v, w \notin fn(P) \cup fn(Q) \cup \{x, y_1, y_2\}$
- **Correctness:** polyadic transition

$$\overline{x}\langle y_1, y_2\rangle.\text{nil} \parallel x(z_1, z_2).R \longrightarrow R[y_1/z_1, y_2/z_2]$$

is mimicked by monadic transition sequence

$$
\begin{array}{lll}
 & \text{new } v\,(\overline{x}\langle v\rangle.\text{nil} \parallel v(w).(\overline{w}\langle y_1\rangle.\text{nil} \parallel v(w).\overline{w}\langle y_2\rangle.\text{nil})) \parallel & \text{(encoding)}\\
 & x(v).\text{new } w\,(\overline{v}\langle w\rangle.\text{nil} \parallel w(z_1).(\overline{v}\langle w\rangle.\text{nil} \parallel w(z_2).R)) & \\
\equiv & \text{new } v\,(\overline{x}\langle v\rangle.\text{nil} \parallel v(w).(\overline{w}\langle y_1\rangle.\text{nil} \parallel v(w).\overline{w}\langle y_2\rangle.\text{nil}) \parallel & \text{(scope extension)}\\
 & x(v).\text{new } w\,(\overline{v}\langle w\rangle.\text{nil} \parallel w(z_1).(\overline{v}\langle w\rangle.\text{nil} \parallel w(z_2).R))) & \\
\longrightarrow & \text{new } v\,(v(w).(\overline{w}\langle y_1\rangle.\text{nil} \parallel v(w).\overline{w}\langle y_2\rangle.\text{nil}) \parallel \text{new } w\,(\overline{v}\langle w\rangle.\text{nil} \parallel w(z_1).(\overline{v}\langle w\rangle.\text{nil} \parallel w(z_2).R))) & \text{(reaction)}\\
\equiv & \text{new } v, w\,(v(w).(\overline{w}\langle y_1\rangle.\text{nil} \parallel v(w).\overline{w}\langle y_2\rangle.\text{nil}) \parallel \overline{v}\langle w\rangle.\text{nil} \parallel w(z_1).(\overline{v}\langle w\rangle.\text{nil} \parallel w(z_2).R)) & \text{(scope extension)}\\
\longrightarrow & \text{new } v, w\,(\overline{w}\langle y_1\rangle.\text{nil} \parallel v(w).\overline{w}\langle y_2\rangle.\text{nil} \parallel w(z_1).(\overline{v}\langle w\rangle.\text{nil} \parallel w(z_2).R)) & \text{(reaction)}\\
\longrightarrow & \text{new } v, w\,(v(w).\overline{w}\langle y_2\rangle.\text{nil} \parallel \overline{v}\langle w\rangle.\text{nil} \parallel w(z_2).R[y_1/z_1]) & \text{(reaction)}\\
\longrightarrow & \text{new } v, w\,(\overline{w}\langle y_2\rangle.\text{nil} \parallel w(z_2).R[y_1/z_1]) & \text{(reaction)}\\
\longrightarrow & \text{new } v, w\; R[y_1/z_1, y_2/z_2] & \text{(reaction)}\\
\equiv & R[y_1/z_1, y_2/z_2] & \text{(congruence)}
\end{array}
$$