

Concurrency Theory

Lecture 2: Calculus of Communicating Systems (CCS)

Joost-Pieter Katoen Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)



{katoen,noll}@cs.rwth-aachen.de

<http://www-i2.informatik.rwth-aachen.de/i2/ct13/>

Winter Semester 2013/14

- 1 Syntax of CCS
- 2 Intuitive Meaning and Examples
- 3 Formal Semantics of CCS
- 4 Process Traces

History:

- Robin Milner: *A Calculus of Communicating Systems*
LNCS 92, Springer, 1980
- Robin Milner: *Communication and Concurrency*
Prentice-Hall, 1989
- Robin Milner: *Communicating and Mobile Systems: the π -calculus*
Cambridge University Press, 1999

Approach: describing parallelism on a **simple and abstract level**, using only a few basic primitives

- no explicit storage (variables)
- no explicit representation of values (numbers, Booleans, ...)

⇒ parallel system reduced to **communication potential**

Definition 2.1 (Syntax of CCS)

- Let A be a set of (action) names.
- $\bar{A} := \{\bar{a} \mid a \in A\}$ denotes the set of co-names.
- $Act := A \cup \bar{A} \cup \{\tau\}$ is the set of actions where τ denotes the silent (or: unobservable) action.
- Let Pid be a set of process identifiers.
- The set Prc of process expressions is defined by the following syntax:

$P ::=$	nil	(inaction)
	$\alpha.P$	(prefixing)
	$P_1 + P_2$	(choice)
	$P_1 \parallel P_2$	(parallel composition)
	$P \setminus L$	(restriction)
	$P[f]$	(relabelling)
	C	(process call)

where $\alpha \in Act$, $L \subseteq A$, $C \in Pid$, and $f : Act \rightarrow Act$ such that $f(\tau) = \tau$ and $f(\bar{a}) = \overline{f(a)}$ for each $a \in A$.

Definition 2.1 (continued)

- A (recursive) process definition is an equation system of the form

$$(C_i = P_i \mid 1 \leq i \leq k)$$

where $k \geq 1$, $C_i \in \text{Pid}$ (pairwise distinct), and $P_i \in \text{Prc}$ (with process identifiers from $\{C_1, \dots, C_k\}$).

Notational Conventions:

- \bar{a} means a
- $\sum_{i=1}^n P_i$ ($n \in \mathbb{N}$) means $P_1 + \dots + P_n$ (where $\sum_{i=1}^0 P_i := \text{nil}$)
- $P \setminus a$ abbreviates $P \setminus \{a\}$
- $[a_1 \mapsto b_1, \dots, a_n \mapsto b_n]$ stands for $f : \text{Act} \rightarrow \text{Act}$ with $f(a_i) = b_i$ ($i \in [n]$) and $f(\alpha) = \alpha$ otherwise
- restriction and relabelling bind stronger than prefixing, prefixing stronger than composition, composition stronger than choice:

$$P \setminus a + b.Q \parallel R \quad \text{means} \quad (P \setminus a) + ((b.Q) \parallel R)$$

- 1 Syntax of CCS
- 2 Intuitive Meaning and Examples
- 3 Formal Semantics of CCS
- 4 Process Traces

Meaning of CCS Constructs

- nil is an **inactive process** that can do nothing.
- $\alpha.P$ can execute α and then behaves as P .
- An action $a \in A$ ($\bar{a} \in \bar{A}$) is interpreted as an **input** (**output**, resp.) operation. Both are complementary: if executed in parallel (i.e., in $P_1 \parallel P_2$), they are merged into a τ -action.
- $P_1 + P_2$ represents the **nondeterministic choice** between P_1 and P_2 .
- $P_1 \parallel P_2$ denotes the **parallel execution** of P_1 and P_2 , involving **interleaving** or **communication**.
- The **restriction** $P \setminus L$ declares each $a \in L$ as a local name which is only known within P .
- The **relabelling** $P[f]$ allows to adapt the naming of actions.
- The behaviour of a **process call** C is given by the right-hand side of the corresponding equation.

Example 2.2

- ① One-place buffer
- ② Two-place buffer
- ③ Parallel specification of two-place buffer

(on the board)

- 1 Syntax of CCS
- 2 Intuitive Meaning and Examples
- 3 Formal Semantics of CCS**
- 4 Process Traces

Labelled Transition Systems

Goal: represent behaviour of system by (infinite) graph

- nodes = system states
- edges = transitions between states

Definition 2.3 (Labelled transition system)

A (**Act**-)labelled transition system (**LTS**) is a triple $(S, Act, \longrightarrow)$ consisting of

- a set S of **states**
- a set Act of (**action**) labels
- a **transition relation** $\longrightarrow \subseteq S \times Act \times S$

For $(s, \alpha, s') \in \longrightarrow$ we write $s \xrightarrow{\alpha} s'$. An LTS is called **finite** if S is so.

Remarks:

- sometimes an **initial state** $s_0 \in S$ is distinguished ($"LTS(s_0)"$)
- (finite) LTSs correspond to (finite) **automata** without final states

We define the assignment

syntax \rightarrow semantics

process definition \mapsto LTS

by induction over the syntactic structure of process expressions. Here we employ **derivation rules** of the form

$$\text{rule name} \frac{\text{premise(s)}}{\text{conclusion}}$$

which can be composed to complete **derivation trees**.

Definition 2.4 (Semantics of CCS)

A process definition $(C_i = P_i \mid 1 \leq i \leq k)$ determines the LTS $(Prc, Act, \longrightarrow)$ whose transitions can be inferred from the following rules $(P, P', Q, Q' \in Prc, \alpha \in Act, \lambda \in A \cup \bar{A}, a \in A)$:

$$\begin{array}{l}
 \text{(Act)} \frac{}{\alpha.P \xrightarrow{\alpha} P} \\
 \text{(Sum}_1\text{)} \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{(Sum}_2\text{)} \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \\
 \text{(Par}_1\text{)} \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad \text{(Par}_2\text{)} \frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'} \\
 \text{(Com)} \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\bar{\lambda}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \qquad \text{(Res)} \frac{P \xrightarrow{\alpha} P' \quad (\alpha, \bar{\alpha} \notin L)}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \\
 \text{(Rel)} \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \text{(Call)} \frac{P \xrightarrow{\alpha} P' \quad (C = P)}{C \xrightarrow{\alpha} P'}
 \end{array}$$

Example 2.5

- ① One-place buffer:

$$B = in.\overline{out}.B$$

- ② Sequential two-place buffer:

$$\begin{aligned}B_0 &= in.B_1 \\ B_1 &= \overline{out}.B_0 + in.B_2 \\ B_2 &= \overline{out}.B_1\end{aligned}$$

- ③ Parallel two-place buffer:

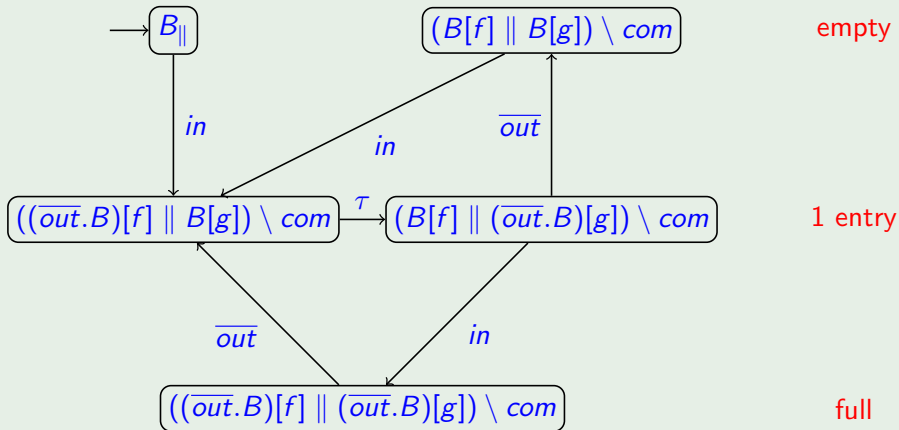
$$\begin{aligned}B_{\parallel} &= (B[f] \parallel B[g]) \setminus com \\ B &= in.\overline{out}.B\end{aligned}$$

where $f := [out \mapsto com]$ and $g := [in \mapsto com]$

(on the board)

Example 2.5 (continued)

Complete LTS of parallel two-place buffer:



- 1 Syntax of CCS
- 2 Intuitive Meaning and Examples
- 3 Formal Semantics of CCS
- 4 Process Traces

Goal: reduce processes to the action sequences they can perform

Definition 2.6 (Trace language)

For every $P \in \text{Prc}$, let

$$\text{Tr}(P) := \{w \in \text{Act}^* \mid \text{ex. } P' \in \text{Prc} \text{ such that } P \xrightarrow{w} P'\}$$

be the **trace language** of P

(where $\xrightarrow{w} := \xrightarrow{a_1} \circ \dots \circ \xrightarrow{a_n}$ for $w = a_1 \dots a_n$).

$P, Q \in \text{Prc}$ are called **trace equivalent** if $\text{Tr}(P) = \text{Tr}(Q)$.

Example 2.7 (One-place buffer)

$$B = in.\overline{out}.B$$

$$\implies \text{Tr}(B) = (in \cdot \overline{out})^* \cdot (in + \epsilon)$$

Remarks:

- The trace language of $P \in \text{Prc}$ is accepted by the LTS of P , interpreted as a (finite or infinite) automaton with **initial state** P and where **every state is final**.
- Trace equivalence is obviously an **equivalence relation** (i.e., reflexive, symmetric, and transitive).
- Trace equivalence identifies processes with **identical LTSs**: the trace language of a process consists of the (finite) paths in the LTS. Thus:

$$LTS(P) = LTS(Q) \implies Tr(P) = Tr(Q)$$