Concurrency Theory

Winter 2025/26

Lecture 2: Calculus of Communicating Systems (CCS)

Thomas Noll, Peter Thiemann
Programming Languages Group
University of Freiburg

https://proglang.github.io/teaching/25ws/ct.html

Thomas Noll, Peter Thiemann

Winter 2025/26

Outline of Lecture 2

The Calculus of Communicating Systems

History

- First development:
  Robin Milner: *A Calculus of Communicating Systems*, LNCS 92, Springer, 1980
- Elaboration and larger case studies:
  Robin Milner: *Communication and Concurrency*, Prentice-Hall, 1989
- Extension to mobile systems:
  Robin Milner: *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press, 1999

The Calculus of Communicating Systems

History

- First development:
  Robin Milner: *A Calculus of Communicating Systems*, LNCS 92, Springer, 1980
- Elaboration and larger case studies:
  Robin Milner: *Communication and Concurrency*, Prentice-Hall, 1989
- Extension to mobile systems:
  Robin Milner: *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press, 1999

Approach

Description of concurrency on a simple and abstract level, using only a few basic primitives

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

$\Rightarrow$ concurrent system reduced to communication potential

Outline of Lecture 2

Syntax of CCS I

Definition 2.1 (Syntax of CCS)

- Let A be a set of (action) names.

Syntax of CCS I

Definition 2.1 (Syntax of CCS)

- Let A be a set of (action) names.
- $\overline{A} := \{\overline{a} \mid a \in A\}$ denotes the set of co-names.

Syntax of CCS I

Definition 2.1 (Syntax of CCS)

- Let A be a set of (action) names.
- $\overline{A} := \{\overline{a} \mid a \in A\}$ denotes the set of co-names.
- $Act := A \cup \overline{A} \cup \{\tau\}$ is the set of actions with the silent (or: unobservable) action τ.

Syntax of CCS I

Definition 2.1 (Syntax of CCS)

- Let A be a set of (action) names.
- $\overline{A} := \{\overline{a} \mid a \in A\}$ denotes the set of co-names.
- $Act := A \cup \overline{A} \cup \{\tau\}$ is the set of actions with the silent (or: unobservable) action τ.
- Let Pid be a set of process identifiers.

Syntax of CCS I

Definition 2.1 (Syntax of CCS)

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

$$
\begin{array}{rll}
P, Q ::= & \mathrm{nil} & \text{(inaction)} \\
\mid & \alpha.P & \text{(prefixing)} \\
\mid & P + Q & \text{(choice)} \\
\mid & P \parallel Q & \text{(parallel composition)} \\
\mid & P \setminus L & \text{(restriction)} \\
\mid & P[f] & \text{(relabelling)} \\
\mid & C & \text{(process call)}
\end{array}
$$

Meaning of CCS Constructs

- nil is an inactive process that can do nothing.

Meaning of CCS Constructs

- nil is an inactive process that can do nothing.
- $\alpha.P$ can execute α and then behaves as P.

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 $a \in A$ ($\overline{a} \in \overline{A}$) is interpreted as an input (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in $P \parallel Q$), they are merged into a τ-action.

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 $a \in A$ ($\overline{a} \in \overline{A}$) is interpreted as an input (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in $P \parallel Q$), they are merged into a τ-action.
- $P + Q$ represents the nondeterministic choice between P and Q.

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 $a \in A$ ($\overline{a} \in \overline{A}$) is interpreted as an input (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in $P \parallel Q$), they are merged into a τ-action.
- $P + Q$ represents the nondeterministic choice between P and Q.
- $P \parallel Q$ denotes the parallel execution of P and Q, involving interleaving or communication.

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 $a \in A$ ($\overline{a} \in \overline{A}$) is interpreted as an input (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in $P \parallel Q$), they are merged into a τ-action.
- $P + Q$ represents the nondeterministic choice between P and Q.
- $P \parallel Q$ denotes the parallel execution of P and Q, involving interleaving or communication.
- The restriction $P \setminus L$ declares each $a \in L$ as a local name which is only known within P.

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 $a \in A$ ($\overline{a} \in \overline{A}$) is interpreted as an input (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in $P \parallel Q$), they are merged into a τ-action.
- $P + Q$ represents the nondeterministic choice between P and Q.
- $P \parallel Q$ denotes the parallel execution of P and Q, 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.

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 $a \in A$ ($\overline{a} \in \overline{A}$) is interpreted as an input (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in $P \parallel Q$), they are merged into a τ-action.
- $P + Q$ represents the nondeterministic choice between P and Q.
- $P \parallel Q$ denotes the parallel execution of P and Q, 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.

Syntax of CCS II

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

Syntax of CCS II

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

Notational Conventions:

- $\overline{\overline{a}}$ means a
- $\sum_{i=1}^{n} P_i$ ($n \in \mathbb{N}$) means $P_1 + \ldots + P_n$ (where $\sum_{i=1}^{0} P_i := \mathrm{nil}$)
- $P \setminus a$ abbreviates $P \setminus \{a\}$
- $[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$ stands for $f : Act \to Act$ with

$$f(a_i) = b_i \text{ for } i \in [n] \quad \text{and} \quad f(\alpha) = \alpha \text{ otherwise}$$

- restriction and relabelling bind stronger than prefixing, prefixing stronger than parallel composition, parallel composition stronger than choice:

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

Outline of Lecture 2

Example 2.2 (Bounded buffers)

(1) One-place buffer:

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

CCS Examples

Example 2.2 (Bounded buffers)

(1) One-place buffer:

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

(2) Two-place buffer:

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

CCS Examples

Example 2.2 (Bounded buffers)

(1) One-place buffer:

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

(2) 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:

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

"Interaction diagram":

Outline of Lecture 2

Labelled Transition Systems

Goal: represent system behaviour by (infinite) graph

- nodes = system states
- edges = transitions between states

Labelled Transition Systems

Goal: represent system behaviour by (infinite) graph

- nodes = system states
- edges = transitions between states

Definition 2.3 (Labelled transition system)

An (*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.

Labelled Transition Systems

Goal: represent system behaviour by (infinite) graph

- nodes = system states
- edges = transitions between states

Definition 2.3 (Labelled transition system)

An (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.

Semantics of CCS I

We define the assignment

$$\begin{array}{rcl} \text{syntax} & \to & \text{semantics} \\ \text{process definition} & \mapsto & \text{LTS} \end{array}$$

by induction over the syntactic structure of process expressions.

Here we employ derivation rules of the form

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

whose instances are composed to form derivation trees (where axioms, i.e., rules without premises, correspond to leaves).

Semantics of CCS II

Reminder: $P, Q ::= \mathrm{nil} \mid \alpha.P \mid P + Q \mid P \parallel Q \mid P \setminus L \mid P[f] \mid C$

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 ($P, P', Q, Q' \in Prc$, $\alpha \in Act$, $\lambda \in A \cup \overline{A}$, $L \subseteq A$, $f : Act \to Act$):

$$\text{(Act)}\,\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{(Sum}_1\text{)}\,\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{(Sum}_2\text{)}\,\frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\text{(Par}_1\text{)}\,\frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad \text{(Par}_2\text{)}\,\frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'} \qquad \text{(Com)}\,\frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

$$\text{(Res)}\,\frac{P \xrightarrow{\alpha} P' \quad (\alpha, \overline{\alpha} \notin L)}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \qquad \text{(Rel)}\,\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \text{(Call)}\,\frac{P \xrightarrow{\alpha} P' \quad (C = P)}{C \xrightarrow{\alpha} P'}$$

Example 2.5 (Bounded buffers; cf. Example 2.2)

(1) One-place buffer: $B = in.\overline{out}.B$

- First step:

$$\text{(Call)}\,\dfrac{\text{(Act)}\,\dfrac{}{in.\overline{out}.B \xrightarrow{in} \overline{out}.B}}{B \xrightarrow{in} \overline{out}.B}$$

Semantics of CCS III

Example 2.5 (Bounded buffers; cf. Example 2.2)

(1) One-place buffer: $B = in.\overline{out}.B$

- First step:

$$\text{(Call)}\,\dfrac{\text{(Act)}\,\dfrac{}{in.\overline{out}.B \xrightarrow{in} \overline{out}.B}}{B \xrightarrow{in} \overline{out}.B}$$

- Second step:

$$\text{(Act)}\,\dfrac{}{\overline{out}.B \xrightarrow{\overline{out}} B}$$

Example 2.5 (Bounded buffers; cf. Example 2.2)

(1) One-place buffer: $B = in.\overline{out}.B$

- First step:

$$(\mathrm{Call})\;\frac{(\mathrm{Act})\;\dfrac{}{in.\overline{out}.B \xrightarrow{in} \overline{out}.B}}{B \xrightarrow{in} \overline{out}.B}$$

- Second step:

$$(\mathrm{Act})\;\frac{}{\overline{out}.B \xrightarrow{\overline{out}} B}$$

$\Rightarrow$ Complete LTS:

$$\begin{tikzcd} \arrow[r] & \boxed{B} \arrow[r, bend left, "in"] & \boxed{\overline{out}.B} \arrow[l, bend left, "\overline{out}"] \end{tikzcd}$$

Example 2.5 (continued)

(2) Sequential two-place buffer:

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

- First step:

$$\text{(Call)}\frac{\text{(Act)}\dfrac{}{in.B_1 \xrightarrow{in} B_1}}{B_0 \xrightarrow{in} B_1}$$

Example 2.5 (continued)

(2) Sequential two-place buffer:
$$\begin{aligned} B_0 &= in.B_1 \\ B_1 &= \overline{out}.B_0 + in.B_2 \\ B_2 &= \overline{out}.B_1 \end{aligned}$$

- First step:

$$\text{(Call)}\,\frac{\text{(Act)}\,\dfrac{}{in.B_1 \xrightarrow{in} B_1}}{B_0 \xrightarrow{in} B_1}$$

- Second step:

$$\text{(Call)}\,\frac{\text{(Sum}_1\text{)}\,\dfrac{\text{(Act)}\,\dfrac{}{\overline{out}.B_0 \xrightarrow{\overline{out}} B_0}}{\overline{out}.B_0 + in.B_2 \xrightarrow{\overline{out}} B_0}}{B_1 \xrightarrow{\overline{out}} B_0}$$

Example 2.5 (continued)

(2) Sequential two-place buffer:
$$\begin{aligned} B_0 &= in.B_1 \\ B_1 &= \overline{out}.B_0 + in.B_2 \\ B_2 &= \overline{out}.B_1 \end{aligned}$$

- First step:

$$\text{(Call)}\ \frac{\text{(Act)}\ \dfrac{}{in.B_1 \xrightarrow{in} B_1}}{B_0 \xrightarrow{in} B_1}$$

- Second step:

$$\text{(Call)}\ \frac{\text{(Sum}_1\text{)}\ \dfrac{\text{(Act)}\ \dfrac{}{\overline{out}.B_0 \xrightarrow{\overline{out}} B_0}}{\overline{out}.B_0 + in.B_2 \xrightarrow{\overline{out}} B_0}}{B_1 \xrightarrow{\overline{out}} B_0}$$

- Like first step: $B_2 \xrightarrow{\overline{out}} B_1$

Example 2.5 (continued)

(2) Sequential two-place buffer:
$$\begin{aligned} B_0 &= in.B_1 \\ B_1 &= \overline{out}.B_0 + in.B_2 \\ B_2 &= \overline{out}.B_1 \end{aligned}$$

- First step:

$$\text{(Call)}\,\frac{\text{(Act)}\,\dfrac{}{in.B_1 \xrightarrow{in} B_1}}{B_0 \xrightarrow{in} B_1}$$

- Second step:

$$\text{(Call)}\,\frac{\text{(Sum}_1\text{)}\,\dfrac{\text{(Act)}\,\dfrac{}{\overline{out}.B_0 \xrightarrow{\overline{out}} B_0}}{\overline{out}.B_0 + in.B_2 \xrightarrow{\overline{out}} B_0}}{B_1 \xrightarrow{\overline{out}} B_0}$$

- Like first step: $B_2 \xrightarrow{\overline{out}} B_1$

- Complete LTS:

$$\begin{tikzcd} \arrow[r] & \boxed{B_0} \arrow[r, bend left, "in"] & \boxed{B_1} \arrow[l, bend left, "\overline{out}"] \arrow[r, bend left, "in"] & \boxed{B_2} \arrow[l, bend left, "\overline{out}"] \end{tikzcd}$$

Semantics of CCS V

Example 2.5 (continued)

(3) Parallel two-place buffer ($f := [out \mapsto com]$, $g := [in \mapsto com]$):

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

First step:

$$(\text{Call})\,\dfrac{(\text{Res})\,\dfrac{(\text{Par}_1)\,\dfrac{(\text{Rel})\,\dfrac{(\text{Call})\,\dfrac{(\text{Act})\,\dfrac{}{in.\overline{out}.B \xrightarrow{in} \overline{out}.B}}{B \xrightarrow{in} \overline{out}.B}}{B[f] \xrightarrow{in} (\overline{out}.B)[f]}}{B[f] \parallel B[g] \xrightarrow{in} (\overline{out}.B)[f] \parallel B[g]}}{(B[f] \parallel B[g]) \setminus com \xrightarrow{in} ((\overline{out}.B)[f] \parallel B[g]) \setminus com}}{B_{\|} \xrightarrow{in} ((\overline{out}.B)[f] \parallel B[g]) \setminus com}$$

Semantics of CCS V

Example 2.5 (continued)

(3) Parallel two-place buffer ($f := [out \mapsto com]$, $g := [in \mapsto com]$):

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

First step:

$$\text{(Call)}\,\dfrac{\text{(Res)}\,\dfrac{\text{(Par}_1\text{)}\,\dfrac{\text{(Rel)}\,\dfrac{\text{(Call)}\,\dfrac{\text{(Act)}\,\dfrac{}{in.\overline{out}.B \xrightarrow{in} \overline{out}.B}}{B \xrightarrow{in} \overline{out}.B}}{B[f] \xrightarrow{in} (\overline{out}.B)[f]}}{B[f] \parallel B[g] \xrightarrow{in} (\overline{out}.B)[f] \parallel B[g]}}{(B[f] \parallel B[g]) \setminus com \xrightarrow{in} ((\overline{out}.B)[f] \parallel B[g]) \setminus com}}{B_{\|} \xrightarrow{in} ((\overline{out}.B)[f] \parallel B[g]) \setminus com}$$

A failing attempt:

$$\text{(Call)}\,\dfrac{\text{(Res)}\,\dfrac{\text{(Par}_1\text{)}\,\dfrac{\text{(Rel)}\,\dfrac{\text{(Call)}\,\dfrac{\text{(Act)}\,\dfrac{}{in.\overline{out}.B \xrightarrow{in} \overline{out}.B}}{B \xrightarrow{in} \overline{out}.B}}{B[g] \xrightarrow{com} (\overline{out}.B)[g]}}{B[f] \parallel B[g] \xrightarrow{com} B[f] \parallel (\overline{out}.B)[g]}}{(B[f] \parallel B[g]) \setminus com \xrightarrow{?} ?}}{B_{\|} \xrightarrow{?} ?}$$

Example 2.5 (continued)

(3) Parallel two-place buffer: $B_{\|} = (B[f] \parallel B[g]) \setminus com \quad (f := [out \mapsto com], g := [in \mapsto com])$
$B = in.\overline{out}.B$

Complete LTS:

$$
\begin{tikzcd}
{} \arrow[r] & B_{\|} \arrow[d, "in"] & & (B[f] \parallel B[g]) \setminus com \arrow[dll, "in"'] & \textcolor{red}{\text{empty}} \\
& ((\overline{out}.B)[f] \parallel B[g]) \setminus com \arrow[rr, "\tau"] & & (B[f] \parallel (\overline{out}.B)[g]) \setminus com \arrow[u, "\overline{out}"] \arrow[dl, "in"] & \textcolor{red}{\text{one entry}} \\
& & ((\overline{out}.B)[f] \parallel (\overline{out}.B)[g]) \setminus com \arrow[ul, "\overline{out}"] & & \textcolor{red}{\text{full}}
\end{tikzcd}
$$

Outline of Lecture 2

The Power of Recursive Definitions

So far: only finite state spaces

The Power of Recursive Definitions

So far: only finite state spaces

Example 2.6 (Counter)

$$C = up.(C \parallel down.\mathrm{nil})$$

The Power of Recursive Definitions

So far: only finite state spaces

Example 2.6 (Counter)

$$C = up.(C \parallel down.\mathrm{nil})$$

gives rise to infinite LTS (abbreviating $down := down.\mathrm{nil}$):

$$\begin{tikzcd}
\arrow[r] & \boxed{C} \arrow[r, "up"] & \boxed{C \parallel down} \arrow[r, "up"] \arrow[d, "down"] & \boxed{C \parallel down \parallel down} \arrow[r, "up"] \arrow[d, "down"] & \boxed{\cdots} \\
& & \boxed{C \parallel \mathrm{nil}} \arrow[r, "up"] & \boxed{\begin{array}{l} C \parallel down \parallel \mathrm{nil} \text{ “=”} \\ C \parallel \mathrm{nil} \parallel down \end{array}} \arrow[r, "up"] \arrow[d, "down"] & \boxed{\cdots} \\
& & & \boxed{C \parallel \mathrm{nil} \parallel \mathrm{nil}} \arrow[r, "up"] & \boxed{\cdots}
\end{tikzcd}$$

The Power of Recursive Definitions

So far: only finite state spaces

Example 2.6 (Counter)

$$C = up.(C \parallel down.\mathrm{nil})$$

gives rise to infinite LTS (abbreviating $down := down.\mathrm{nil}$):

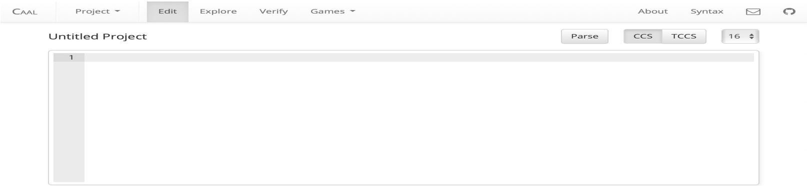
$$\begin{tikzcd}
{} \arrow[r] & \boxed{C} \arrow[r, "up"] & \boxed{C \parallel down} \arrow[r, "up"] \arrow[d, "down"] & \boxed{C \parallel down \parallel down} \arrow[r, "up"] \arrow[d, "down"] & \boxed{\cdots} \\
 & & \boxed{C \parallel \mathrm{nil}} \arrow[r, "up"] & \boxed{\begin{matrix} C \parallel down \parallel \mathrm{nil} \text{ ``=''} \\ C \parallel \mathrm{nil} \parallel down \end{matrix}} \arrow[r, "up"] \arrow[d, "down"] & \boxed{\cdots} \\
 & & & \boxed{C \parallel \mathrm{nil} \parallel \mathrm{nil}} \arrow[r, "up"] & \boxed{\cdots}
\end{tikzcd}$$

Sequential "specification":

$$C_0 = up.C_1$$
$$C_n = up.C_{n+1} + down.C_{n-1} \qquad (n > 0)$$

Outline of Lecture 2

The CAAL Tool



CAAL (Concurrency Workbench, Aalborg Edition; https://caal.cs.aau.dk/)

- Smart editor
- Visualisation of generated LTS
- Equivalence checking w.r.t. several bisimulation, simulation and trace equivalences
- Model checking of (recursive) HML formulae
- Generation of distinguishing formulae for non-equivalent processes
- (Bi)simulation and model checking games

Outline of Lecture 2

Summary: CCS

Summary

- Process behaviour defined by (synchronising) actions
- Syntax given by recursive definitions of processes
  - inaction nil
  - prefixing $\alpha.P$
  - choice $P + Q$
  - parallel composition $P \parallel Q$
  - restriction $P \setminus L$
  - relabelling $P[f]$
- Semantics given by (finite or infinite) labelled transition system
- Implemented by CAAL Tool