Concurrency Theory

Winter 2025/26

Lecture 12: Modelling Mutual-Exclusion Algorithms & Value-Passing 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 12

Peterson's Mutual Exclusion Algorithm

- **Goal:** ensuring exclusive access to non-shared resources
- Here: two competing processes P_1, P_2 and shared variables
  - b_1, b_2 (Boolean, initially false) – b_i indicates that P_i wants to enter critical section
  - k (in $\{1, 2\}$, arbitrary initial value) – index of prioritised process ("turn variable")
- P_i uses local variable $j := 2 - i$ (index of other process)

Peterson's Mutual Exclusion Algorithm

- **Goal:** ensuring exclusive access to non-shared resources
- Here: two competing processes P_1, P_2 and shared variables
  - b_1, b_2 (Boolean, initially false) – b_i indicates that P_i wants to enter critical section
  - k (in $\{1, 2\}$, arbitrary initial value) – index of prioritised process ("turn variable")
- P_i uses local variable $j := 2 - i$ (index of other process)

Algorithm 12.1 (Peterson's algorithm for P_i)

while true do
  "non-critical section";
  $b_i :=$ true;
  $k := j$;
  await $\neg b_j \vee k = i$;
  "critical section";
  $b_i :=$ false;
end

Representing Shared Variables in CCS

- Not directly expressible in CCS (communication by handshaking)
- Idea: consider variables as processes that communicate with environment by processing read/write requests

Representing Shared Variables in CCS

- Not directly expressible in CCS (communication by handshaking)
- Idea: consider variables as processes that communicate with environment by processing read/write requests

Example 12.2 (Shared variables in Peterson's algorithm)

- Encoding of b_1 with two (process) states B_{1t} (value tt) and B_{1f} (ff)
- Read access along ports $b1rt$ (in state B_{1t}) and $b1rf$ (in state B_{1f})
- Write access along ports $b1wt$ and $b1wf$ (in both states)

Representing Shared Variables in CCS

- Not directly expressible in CCS (communication by handshaking)
- Idea: consider variables as processes that communicate with environment by processing read/write requests

Example 12.2 (Shared variables in Peterson's algorithm)

- Encoding of b_1 with two (process) states B_{1t} (value tt) and B_{1f} (ff)
- Read access along ports $b1rt$ (in state B_{1t}) and $b1rf$ (in state B_{1f})
- Write access along ports $b1wt$ and $b1wf$ (in both states)
- Possible behaviours:

$$B_{1f} = \overline{b1rf}.B_{1f} + b1wf.B_{1f} + b1wt.B_{1t}$$
$$B_{1t} = \overline{b1rt}.B_{1t} + b1wf.B_{1f} + b1wt.B_{1t}$$

Representing Shared Variables in CCS

- Not directly expressible in CCS (communication by handshaking)
- Idea: consider variables as processes that communicate with environment by processing read/write requests

Example 12.2 (Shared variables in Peterson's algorithm)

- Encoding of b_1 with two (process) states B_{1t} (value tt) and B_{1f} (ff)
- Read access along ports $b1rt$ (in state B_{1t}) and $b1rf$ (in state B_{1f})
- Write access along ports $b1wt$ and $b1wf$ (in both states)
- Possible behaviours:

$$
\begin{aligned}
B_{1f} &= \overline{b1rf}.B_{1f} + b1wf.B_{1f} + b1wt.B_{1t} \\
B_{1t} &= \overline{b1rt}.B_{1t} + b1wf.B_{1f} + b1wt.B_{1t}
\end{aligned}
$$

- Similarly for b_2 and k:

$$
\begin{aligned}
B_{2f} &= \overline{b2rf}.B_{2f} + b2wf.B_{2f} + b2wt.B_{2t} \\
B_{2t} &= \overline{b2rt}.B_{2t} + b2wf.B_{2f} + b2wt.B_{2t}
\end{aligned}
$$

$$
\begin{aligned}
K_1 &= \overline{kr1}.K_1 + kw1.K_1 + kw2.K_2 \\
K_2 &= \overline{kr2}.K_2 + kw1.K_1 + kw2.K_2
\end{aligned}
$$

Modelling the Processes in CCS

Assumption: P_i cannot fail or terminate within critical section

Peterson's algorithm

while true do
  "non-critical section";
  b_i := true;
  k := j;
  await $\neg b_j \vee k = i$;
  "critical section";
  b_i := false;
end

Modelling the Processes in CCS

Assumption: P_i cannot fail or terminate within critical section

Peterson's algorithm

while true do
  "non-critical section";
  b_i := true;
  $k := j$;
  await $\neg b_j \vee k = i$;
  "critical section";
  b_i := false;
end

CCS representation

$$
\begin{aligned}
P_1 &= \overline{b1wt}.\overline{kw2}.P_{11} \\
P_{11} &= b2rf.P_{12} + \\
& b2rt.(kr1.P_{12} + kr2.P_{11}) \\
P_{12} &= enter1.exit1.\overline{b1wf}.P_1
\end{aligned}
$$

$$
\begin{aligned}
P_2 &= \overline{b2wt}.\overline{kw1}.P_{21} \\
P_{21} &= b1rf.P_{22} + \\
& b1rt.(kr1.P_{21} + kr2.P_{22}) \\
P_{22} &= enter2.exit2.\overline{b2wf}.P_2
\end{aligned}
$$

$$Peterson = (P_1 \parallel P_2 \parallel B_{1f} \parallel B_{2f} \parallel K_1) \setminus L$$

$$
\text{for } L = \{b1rf, b1rt, b1wf, b1wt, b2rf, b2rt, b2wf, b2wt, kr1, kr2, kw1, kw2\}
$$

Modelling the Processes in CCS

Assumption: P_i cannot fail or terminate within critical section

Peterson's algorithm

while true do
  "non-critical section";
  b_i := true;
  $k := j$;
  await $\neg b_j \vee k = i$;
  "critical section";
  b_i := false;
end

CCS representation

$$\begin{aligned}
P_1 &= \overline{b1wt}.\overline{kw2}.P_{11}\\
P_{11} &= b2rf.P_{12} + {}\\
&\quad\; b2rt.(kr1.P_{12} + kr2.P_{11})\\
P_{12} &= enter1.exit1.\overline{b1wf}.P_1\\[1ex]
P_2 &= \overline{b2wt}.\overline{kw1}.P_{21}\\
P_{21} &= b1rf.P_{22} + {}\\
&\quad\; b1rt.(kr1.P_{21} + kr2.P_{22})\\
P_{22} &= enter2.exit2.\overline{b2wf}.P_2\\[1ex]
Peterson &= (P_1 \parallel P_2 \parallel B_{1f} \parallel B_{2f} \parallel K_1) \setminus L\\
\text{for } L &= \{b1rf, b1rt, b1wf, b1wt,\\
&\quad\; b2rf, b2rt, b2wf, b2wt,\\
&\quad\; kr1, kr2, kw1, kw2\}
\end{aligned}$$

Modelling the Processes in CCS

Assumption: P_i cannot fail or terminate within critical section

Peterson's algorithm

while true do
  "non-critical section";
  b_i := true;
  $k := j$;
  await $\neg b_j \lor k = i$;
  "critical section";
  b_i := false;
end

CCS representation

$$\begin{aligned}
P_1 &= \overline{b1wt}.\overline{kw2}.P_{11} \\
P_{11} &= b2rf.P_{12} + \\
&\quad\; b2rt.(kr1.P_{12} + kr2.P_{11}) \\
P_{12} &= enter1.exit1.\overline{b1wf}.P_1
\end{aligned}$$

$$\begin{aligned}
P_2 &= \overline{b2wt}.\overline{kw1}.P_{21} \\
P_{21} &= b1rf.P_{22} + \\
&\quad\; b1rt.(kr1.P_{21} + kr2.P_{22}) \\
P_{22} &= enter2.exit2.\overline{b2wf}.P_2
\end{aligned}$$

$$Peterson = (P_1 \parallel P_2 \parallel B_{1f} \parallel B_{2f} \parallel K_1) \setminus L$$

$$\text{for } L = \{b1rf, b1rt, b1wf, b1wt, b2rf, b2rt, b2wf, b2wt, kr1, kr2, kw1, kw2\}$$

Modelling the Processes in CCS

Assumption: P_i cannot fail or terminate within critical section

Peterson's algorithm

while true do
  "non-critical section";
  b_i := true;
  $k := j$;
  await $\neg b_j \vee k = i$;
  "critical section";
  b_i := false;
end

CCS representation

$$
\begin{aligned}
P_1 &= \overline{b1wt}.\overline{kw2}.P_{11} \\
P_{11} &= b2rf.P_{12} + \\
&\quad\; b2rt.(kr1.P_{12} + kr2.P_{11}) \\
P_{12} &= enter1.exit1.\overline{b1wf}.P_1 \\[1ex]
P_2 &= \overline{b2wt}.\overline{kw1}.P_{21} \\
P_{21} &= b1rf.P_{22} + \\
&\quad\; b1rt.(kr1.P_{21} + kr2.P_{22}) \\
P_{22} &= enter2.exit2.\overline{b2wf}.P_2
\end{aligned}
$$

$$Peterson = (P_1 \parallel P_2 \parallel B_{1f} \parallel B_{2f} \parallel K_1) \setminus L$$

for $L = \{b1rf, b1rt, b1wf, b1wt,$
$b2rf, b2rt, b2wf, b2wt,$
$kr1, kr2, kw1, kw2\}$

Modelling the Processes in CCS

Assumption: P_i cannot fail or terminate within critical section

Peterson's algorithm

while true do
  "non-critical section";
  b_i := true;
  $k := j$;
  await $\neg b_j \vee k = i$;
  "critical section";
  b_i := false;
end

CCS representation

$$\begin{aligned}
P_1 &= \overline{b1wt}.\overline{kw2}.P_{11} \\
P_{11} &= b2rf.P_{12} + \\
& b2rt.(kr1.P_{12} + kr2.P_{11}) \\
P_{12} &= enter1.exit1.\overline{b1wf}.P_1 \\
P_2 &= \overline{b2wt}.\overline{kw1}.P_{21} \\
P_{21} &= b1rf.P_{22} + \\
& b1rt.(kr1.P_{21} + kr2.P_{22}) \\
P_{22} &= enter2.exit2.\overline{b2wf}.P_2
\end{aligned}$$

$$Peterson = (P_1 \parallel P_2 \parallel B_{1f} \parallel B_{2f} \parallel K_1) \setminus L$$

for $L = \{b1rf, b1rt, b1wf, b1wt, b2rf, b2rt, b2wf, b2wt, kr1, kr2, kw1, kw2\}$

Modelling the Processes in CCS

Assumption: P_i cannot fail or terminate within critical section

Peterson's algorithm

while true do
  "non-critical section";
  b_i := true;
  $k := j$;
  await $\neg b_j \vee k = i$;
  "critical section";
  b_i := false;
end

CCS representation

$$\begin{aligned}
P_1 &= \overline{b1wt}.\overline{kw2}.P_{11}\\
P_{11} &= b2rf.P_{12} + \\
&\quad\ b2rt.(kr1.P_{12} + kr2.P_{11})\\
P_{12} &= enter1.exit1.\overline{b1wf}.P_1\\
P_2 &= \overline{b2wt}.\overline{kw1}.P_{21}\\
P_{21} &= b1rf.P_{22} + \\
&\quad\ b1rt.(kr1.P_{21} + kr2.P_{22})\\
P_{22} &= enter2.exit2.\overline{b2wf}.P_2
\end{aligned}$$

$$Peterson = (P_1 \parallel P_2 \parallel B_{1f} \parallel B_{2f} \parallel K_1) \setminus L$$

$$\text{for } L = \{b1rf, b1rt, b1wf, b1wt, b2rf, b2rt, b2wf, b2wt, kr1, kr2, kw1, kw2\}$$

Modelling the Processes in CCS

Assumption: P_i cannot fail or terminate within critical section

Peterson's algorithm

while true do
  "non-critical section";
  $b_i :=$ true;
  $k := j$;
  await $\neg b_j \lor k = i$;
  "critical section";
  $b_i :=$ false;
end

CCS representation

$$
\begin{aligned}
P_1 &= \overline{b1wt}.\overline{kw2}.P_{11}\\
P_{11} &= b2rf.P_{12} + \\
&\quad\ b2rt.(kr1.P_{12} + kr2.P_{11})\\
P_{12} &= enter1.exit1.\overline{b1wf}.P_1\\
P_2 &= \overline{b2wt}.\overline{kw1}.P_{21}\\
P_{21} &= b1rf.P_{22} + \\
&\quad\ b1rt.(kr1.P_{21} + kr2.P_{22})\\
P_{22} &= enter2.exit2.\overline{b2wf}.P_2
\end{aligned}
$$

$$
Peterson = (P_1 \parallel P_2 \parallel B_{1f} \parallel B_{2f} \parallel K_1) \setminus L
$$

for $L = \{b1rf, b1rt, b1wf, b1wt,$
$b2rf, b2rt, b2wf, b2wt,$
$kr1, kr2, kw1, kw2\}$

Outline of Lecture 12

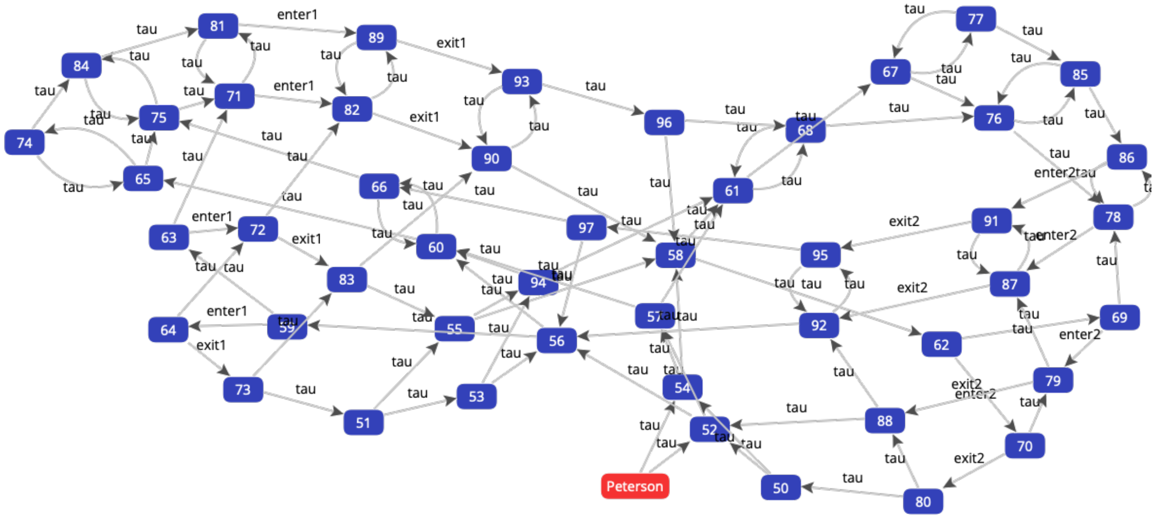
Specifying the Algorithm in CAAL

CAAL   Project   Edit   Explore   Verify   Games   About   Syntax

MutEx   Parse   CCS   TCCS   14

```
*
* Peterson's algorithm for mutual exclusion (Lecture 12)
*

B1f = 'b1rf.B1f + b1wf.B1f + b1wt.B1t;
* B1f = 'b1rf.B1f + b1wf.B1f + b1wt.enabled1.B1t;
B1t = 'b1rt.B1t + b1wf.B1f + b1wt.B1t;

B2f = 'b2rf.B2f + b2wf.B2f + b2wt.B2t;
* B2f = 'b2rf.B2f + b2wf.B2f + b2wt.enabled2.B2t;
B2t = 'b2rt.B2t + b2wf.B2f + b2wt.B2t;

K1 = 'kr1.K1 + kw1.K1 + kw2.K2;
K2 = 'kr2.K2 + kw1.K1 + kw2.K2;

P1 = 'b1wt.'kw2.P11;
P11 = b2rf.P12 + b2rt.(kr2.P11 + kr1.P12);
P12 = enter1.exit1.'b1wf.P1;

P2 = 'b2wt.'kw1.P21;
P21 = b1rf.P22 + b1rt.(kr1.P21 + kr2.P22);
P22 = enter2.exit2.'b2wf.P2;

set L = {b1rf, b2rf, b1rt, b2rt, b1wf, b2wf, b1wt, b2wt, kr1, kr2, kw1, kw2};
Peterson = (P1 | P2 | B1f | B2f | K1) \ L;

MutExCCS = enter1.exit1.MutExCCS + enter2.exit2.MutExCCS;
```

Obtaining the LTS Using CAAL



Outline of Lecture 12

The Mutual Exclusion Property

- **Done:** Formal description of Peterson's algorithm
- **To do:** Analysing its behaviour (manually or with tool support)
- **Question:** What does "ensuring mutual exclusion" formally mean?

The Mutual Exclusion Property

- **Done:** Formal description of Peterson's algorithm
- **To do:** Analysing its behaviour (manually or with tool support)
- **Question:** What does "ensuring mutual exclusion" formally mean?

Mutual exclusion

At no point in the execution of the algorithm, processes P_1 and P_2 will both be in their critical section.

Equivalently:
It is always the case that either P_1 or P_2 or both are not in their critical section.

Model Checking Mutual Exclusion in HML

Mutual exclusion

It is always the case that either P_1 or P_2 or both are not in their critical section.

Model Checking Mutual Exclusion in HML

Mutual exclusion

It is always the case that either P_1 or P_2 or both are not in their critical section.

Observations:

- Mutual exclusion is an invariance property (“always”).
- P_i is in its critical section iff action *exit i* is enabled.

Model Checking Mutual Exclusion in HML

Mutual exclusion

It is always the case that either P_1 or P_2 or both are not in their critical section.

Observations:

- Mutual exclusion is an invariance property (“always”).
- P_i is in its critical section iff action *exit i* is enabled.

Mutual exclusion in HML

$$Peterson \models MutExHML$$

$$\begin{aligned} MutExHML &:= Inv(NotBoth) \\ Inv(F) &\stackrel{max}{=} F \wedge [Act]Inv(F) \qquad \text{(cf. Theorem 11.1)} \\ NotBoth &:= [exit1]\mathrm{ff} \vee [exit2]\mathrm{ff} \end{aligned}$$

Absence of Deadlocks

Absence of deadlocks

It is always the case that the system can progress, i.e., perform any action.

Absence of Deadlocks

Absence of deadlocks

It is always the case that the system can progress, i.e., perform any action.

Absence of deadlocks in HML

$$Peterson \models NoDeadlock$$

$$\begin{aligned} NoDeadlock &:= Inv(CanProgress) \\ Inv(F) &\stackrel{max}{=} F \wedge [Act]Inv(F) \\ CanProgress &:= \langle Act \rangle \mathrm{tt} \end{aligned}$$

Possibility of Livelocks

Possibility of livelocks

It is possible for the system to reach a state which has a livelock (i.e., an infinite sequence of internal steps).

Possibility of Livelocks

Possibility of livelocks

It is possible for the system to reach a state which has a livelock (i.e., an infinite sequence of internal steps).

Possibility of livelocks in HML (cf. Example 11.11)

$$\mathit{Peterson} \models \mathit{HasLivelock}$$

$$\begin{aligned} \mathit{HasLivelock} &:= \mathit{Pos}(\mathit{Livelock}) \\ \mathit{Pos}(F) &\stackrel{\mathit{min}}{=} F \vee \langle \mathit{Act} \rangle \mathit{Pos}(F) \qquad \text{(cf. Theorem 11.2)} \\ \mathit{Livelock} &\stackrel{\mathit{max}}{=} \langle \tau \rangle \mathit{Livelock} \end{aligned}$$

Outline of Lecture 12

Verification by Bisimulation Checking

- **Goal:** express desired behaviour of algorithm as an “abstract” CCS process

Verification by Bisimulation Checking

- **Goal:** express desired behaviour of algorithm as an “abstract” CCS process
- Intuitively:
  (1) Initially, either P_1 or P_2 can enter its critical section.
  (2) Afterwards, the other process cannot enter the critical section before the first has left.

Verification by Bisimulation Checking

- **Goal:** express desired behaviour of algorithm as an "abstract" CCS process
- Intuitively:
  (1) Initially, either P_1 or P_2 can enter its critical section.
  (2) Afterwards, the other process cannot enter the critical section before the first has left.

Mutual exclusion in CCS

$$\mathit{MutExCCS} = \mathit{enter1}.\mathit{exit1}.\mathit{MutExCCS} + \mathit{enter2}.\mathit{exit2}.\mathit{MutExCCS}$$

Verification by Bisimulation Checking

- **Goal:** express desired behaviour of algorithm as an “abstract” CCS process
- Intuitively:
  (1) Initially, either P_1 or P_2 can enter its critical section.
  (2) Afterwards, the other process cannot enter the critical section before the first has left.

Mutual exclusion in CCS

$$MutExCCS = enter1.exit1.MutExCCS + enter2.exit2.MutExCCS$$

- Weak bisimilarity (Definition 5.10) does *not* hold as *Peterson* satisfies additional fairness constraints (prioritisation via variable k — distinguishing formula $\langle\langle\tau\rangle\rangle[[enter2]]\mathrm{ff}$):

$$Peterson \not\approx MutExCCS.$$

Verification by Bisimulation Checking

- **Goal:** express desired behaviour of algorithm as an "abstract" CCS process
- Intuitively:
  (1) Initially, either P_1 or P_2 can enter its critical section.
  (2) Afterwards, the other process cannot enter the critical section before the first has left.

Mutual exclusion in CCS

$$\mathit{MutExCCS} = \mathit{enter1}.\mathit{exit1}.\mathit{MutExCCS} + \mathit{enter2}.\mathit{exit2}.\mathit{MutExCCS}$$

- Weak bisimilarity (Definition 5.10) does *not* hold as *Peterson* satisfies additional fairness constraints (prioritisation via variable k — distinguishing formula $\langle\langle\tau\rangle\rangle[[\mathit{enter2}]]\mathrm{ff}$):

$$\mathit{Peterson} \not\approx \mathit{MutExCCS}.$$

- However, *Peterson* and *MutExCCS* are weakly simulation equivalent:

$$\mathit{Peterson} \sqsubseteq\!\!\!\sim \mathit{MutExCCS} \qquad \text{and} \qquad \mathit{MutExCCS} \sqsubseteq\!\!\!\sim \mathit{Peterson}$$

  where $P \sqsubseteq\!\!\!\sim Q$ denotes that Q weakly simulates P, i.e., can respond to every $\xrightarrow{\alpha}$-step of P by performing a $\stackrel{\alpha}{\Longrightarrow}$-step (cf. Definition 5.3 of strong simulation).

Verification by Bisimulation Checking

- **Goal:** express desired behaviour of algorithm as an “abstract” CCS process
- Intuitively:
  (1) Initially, either P_1 or P_2 can enter its critical section.
  (2) Afterwards, the other process cannot enter the critical section before the first has left.

Mutual exclusion in CCS

$$\mathit{MutExCCS} = \mathit{enter1}.\mathit{exit1}.\mathit{MutExCCS} + \mathit{enter2}.\mathit{exit2}.\mathit{MutExCCS}$$

- Weak bisimilarity (Definition 5.10) does *not* hold as *Peterson* satisfies additional fairness constraints (prioritisation via variable k — distinguishing formula $\langle\langle\tau\rangle\rangle[[\mathit{enter2}]]\mathrm{ff}$):

$$\mathit{Peterson} \not\approx \mathit{MutExCCS}.$$

- However, *Peterson* and *MutExCCS* are weakly simulation equivalent:

$$\mathit{Peterson} \sqsubseteq_{\approx} \mathit{MutExCCS} \qquad \text{and} \qquad \mathit{MutExCCS} \sqsubseteq_{\approx} \mathit{Peterson}$$

  where $P \sqsubseteq_{\approx} Q$ denotes that Q weakly simulates P, i.e., can respond to every $\xrightarrow{\alpha}$-step of P by performing a $\xRightarrow{\alpha}$-step (cf. Definition 5.3 of strong simulation).
- In particular, this implies that *Peterson* and *MutExCCS* are observationally trace equivalent (Definition 6.6):

$$\mathit{ObsTr}(\mathit{Peterson}) = \mathit{ObsTr}(\mathit{MutExCCS}).$$

Outline of Lecture 12

Verification by Testing I

Approach:

- Make mutual exclusion algorithm interact with “monitor” process that observes its behaviour.
- Report error if and when undesired behaviour is detected.

Verification by Testing I

Approach:

- Make mutual exclusion algorithm interact with "monitor" process that observes its behaviour.
- Report error if and when undesired behaviour is detected.

The monitor process

$$
\begin{aligned}
\mathit{MutExTest} &:= \overline{\mathit{enter1}}.\mathit{MutExTest}_1 + \overline{\mathit{enter2}}.\mathit{MutExTest}_2 \\
\mathit{MutExTest}_1 &:= \overline{\mathit{exit1}}.\mathit{MutExTest} + \overline{\mathit{enter2}}.\overline{\mathit{bad}}.\mathrm{nil} \\
\mathit{MutExTest}_2 &:= \overline{\mathit{exit2}}.\mathit{MutExTest} + \overline{\mathit{enter1}}.\overline{\mathit{bad}}.\mathrm{nil}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Test} &:= (\mathit{Peterson} \parallel \mathit{MutExTest}) \setminus L' \\
L' &:= \{\mathit{enter1}, \mathit{enter2}, \mathit{exit1}, \mathit{exit2}\}
\end{aligned}
$$

Verification by Testing II

Lemma 12.2

Let $P \in Prc$ *be a process whose only visible actions are contained in* L'. *Then*

$$Test \stackrel{\overline{bad}}{\Longrightarrow} \quad \text{iff} \quad \text{either } P \stackrel{\sigma}{\Longrightarrow} \stackrel{enter1}{\Longrightarrow} \stackrel{enter2}{\Longrightarrow} \text{ or } P \stackrel{\sigma}{\Longrightarrow} \stackrel{enter2}{\Longrightarrow} \stackrel{enter1}{\Longrightarrow}$$

for some sequence of actions $\sigma \in (enter1\ exit1 \mid enter2\ exit2)^*$.

Verification by Testing II

Lemma 12.2

Let $P \in Prc$ be a process whose only visible actions are contained in L'. Then

$$Test \stackrel{\overline{bad}}{\Longrightarrow} \quad \text{iff} \quad \text{either } P \stackrel{\sigma}{\Longrightarrow} \stackrel{enter1}{\Longrightarrow} \stackrel{enter2}{\Longrightarrow} \text{ or } P \stackrel{\sigma}{\Longrightarrow} \stackrel{enter2}{\Longrightarrow} \stackrel{enter1}{\Longrightarrow}$$

for some sequence of actions $\sigma \in (enter1\ exit1 \mid enter2\ exit2)^*$.

Proof.

see Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen and Jiří Srba: *Reactive Systems: Modelling, Specification and Verification*, Cambridge University Press, 2007, Proposition 7.2 □

Verification by Testing II

Lemma 12.2

Let $P \in Prc$ be a process whose only visible actions are contained in L'. Then

$$Test \stackrel{\overline{bad}}{\Longrightarrow} \quad \text{iff} \quad \text{either } P \stackrel{\sigma}{\Longrightarrow} \stackrel{enter1}{\Longrightarrow} \stackrel{enter2}{\Longrightarrow} \text{ or } P \stackrel{\sigma}{\Longrightarrow} \stackrel{enter2}{\Longrightarrow} \stackrel{enter1}{\Longrightarrow}$$

for some sequence of actions $\sigma \in (enter1\ exit1 \mid enter2\ exit2)^*$.

Proof.

see Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen and Jiří Srba: *Reactive Systems: Modelling, Specification and Verification*, Cambridge University Press, 2007, Proposition 7.2 □

Absence of bad transitions

$$Test \models NoBadTransition$$

$$NoBadTransition := Inv([\overline{bad}]\mathrm{ff})$$

$$Inv(F) \stackrel{max}{=} F \wedge [Act]Inv(F)$$

Outline of Lecture 12

Value-Passing CCS

- **So far:** pure CCS
  - communication = mere synchronisation
  - no (explicit) exchange of data
- **But:** processes usually do pass around data

Value-Passing CCS

- **So far:** pure CCS
  - communication = mere synchronisation
  - no (explicit) exchange of data
- **But:** processes usually do pass around data

$\Rightarrow$ Value-passing CCS

- Introduced in Robin Milner: *Communication and Concurrency*, Prentice-Hall, 1989
- Assumption (for simplicity): only integers as data type

Value-Passing CCS

- **So far:** pure CCS
  - communication = mere synchronisation
  - no (explicit) exchange of data
- **But:** processes usually do pass around data

$\Rightarrow$ Value-passing CCS

- Introduced in Robin Milner: *Communication and Concurrency*, Prentice-Hall, 1989
- Assumption (for simplicity): only integers as data type

Example 12.3 (One-place buffer with data; cf. Example 2.5)

One-place buffer that outputs successor of stored value:

$$B = in(x).B'(x)$$
$$B'(x) = \overline{out}(x+1).B$$

Syntax of Value-Passing CCS I

Definition 12.4 (Syntax of value-passing CCS)

- Let A, $\overline{A}$, Pid (ranked) as in Definition 2.1.

Definition 12.4 (Syntax of value-passing CCS)

- Let A, $\overline{A}$, Pid (ranked) as in Definition 2.1.
- Let e and b be integer and Boolean expressions, resp., built from integer variables $x, y, \ldots$

Syntax of Value-Passing CCS I

Definition 12.4 (Syntax of value-passing CCS)

- Let A, $\overline{A}$, Pid (ranked) as in Definition 2.1.
- Let e and b be integer and Boolean expressions, resp., built from integer variables $x, y, \ldots$
- The set Prc^+ of value-passing process expressions is defined by:

$$
\begin{array}{rlll}
P ::= & \mathrm{nil} & & \text{(inaction)} \\
| & a(x).P & & \text{(input prefixing)} \\
| & \overline{a}(e).P & & \text{(output prefixing)} \\
| & \tau.P & & \text{(τ prefixing)} \\
| & P_1 + P_2 & & \text{(choice)} \\
| & P_1 \parallel P_2 & & \text{(parallel composition)} \\
| & P \setminus L & & \text{(restriction)} \\
| & P[f] & & \text{(relabelling)} \\
| & \text{if } b \text{ then } P & & \text{(conditional)} \\
| & C(e_1, \ldots, e_n) & & \text{(process call)}
\end{array}
$$

where $a \in A$, $L \subseteq A$, $C \in Pid$ (of rank $n \in \mathbb{N}$), and $f : A \to A$.

Syntax of Value-Passing CCS II

Definition 12.4 (Syntax of value-passing CCS; continued)

A value-passing process definition is an equation system of the form

$$(C_i(x_1, \ldots, x_{n_i}) = P_i \mid 1 \leq i \leq k)$$

where

- $k \geq 1$,
- $C_i \in Pid$ of rank n_i (pairwise distinct),
- $P_i \in Prc^+$ (with process identifiers from $\{C_1, \ldots, C_k\}$), and
- all occurrences of an integer variable y in each P_i are bound, i.e., $y \in \{x_1, \ldots, x_{n_i}\}$ or y is in the scope of an input prefix of the form $a(y)$ (to ensure well-definedness of values).

Syntax of Value-Passing CCS II

Definition 12.4 (Syntax of value-passing CCS; continued)

A value-passing process definition is an equation system of the form

$$(C_i(x_1, \ldots, x_{n_i}) = P_i \mid 1 \leq i \leq k)$$

where

- $k \geq 1$,
- $C_i \in Pid$ of rank n_i (pairwise distinct),
- $P_i \in Prc^+$ (with process identifiers from $\{C_1, \ldots, C_k\}$), and
- all occurrences of an integer variable y in each P_i are bound, i.e., $y \in \{x_1, \ldots, x_{n_i}\}$ or y is in the scope of an input prefix of the form $a(y)$ (to ensure well-definedness of values).

Example 12.5

(1) $C(x) = \overline{a}(x+1).b(y).C(y)$ is allowed

Syntax of Value-Passing CCS II

Definition 12.4 (Syntax of value-passing CCS; continued)

A value-passing process definition is an equation system of the form

$$(C_i(x_1, \ldots, x_{n_i}) = P_i \mid 1 \leq i \leq k)$$

where

- $k \geq 1$,
- $C_i \in Pid$ of rank n_i (pairwise distinct),
- $P_i \in Prc^+$ (with process identifiers from $\{C_1, \ldots, C_k\}$), and
- all occurrences of an integer variable y in each P_i are bound, i.e., $y \in \{x_1, \ldots, x_{n_i}\}$ or y is in the scope of an input prefix of the form $a(y)$ (to ensure well-definedness of values).

Example 12.5

(1) $C(x) = \overline{a}(x + 1).b(y).C(y)$ is allowed

(2) $C(x) = \overline{a}(x + 1).\overline{a}(y + 2).\mathrm{nil}$ is disallowed as y is not bound

Outline of Lecture 12

Definition 12.6 (Semantics of value-passing CCS)

A value-passing process definition $(C_i(x_1, \ldots, x_{n_i}) = P_i \mid 1 \leq i \leq k)$ determines the LTS $(Prc^+, Act, \longrightarrow)$ with $Act := (A \cup \overline{A}) \times \mathbb{Z} \cup \{\tau\}$ whose transitions can be inferred from the following rules ($P, P', Q, Q' \in Prc^+$, $a \in A$, x_i integer variables, e_i/b integer/Boolean expressions, $z \in \mathbb{Z}$, $\alpha \in Act$, $\lambda \in (A \cup \overline{A}) \times \mathbb{Z}$):

$$\text{(In)}\ \frac{}{a(x).P \xrightarrow{a(z)} P[z/x]} \qquad \text{(Out)}\ \frac{(z \text{ value of } e)}{\overline{a}(e).P \xrightarrow{\overline{a}(z)} P} \qquad \text{(Tau)}\ \frac{}{\tau.P \xrightarrow{\tau} P}$$

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

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

Semantics of Value-Passing CCS II

Definition 12.6 (Semantics of value-passing CCS; continued)

$$\text{(Rel)}\ \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \text{(Res)}\ \frac{P \xrightarrow{\alpha} P' \quad (\alpha \notin (L \cup \overline{L}) \times \mathbb{Z})}{P \setminus L \xrightarrow{\alpha} P' \setminus L}$$

$$\text{(If)}\ \frac{P \xrightarrow{\alpha} P' \quad (b \text{ true})}{\text{if } b \text{ then } P \xrightarrow{\alpha} P'} \qquad \text{(Call)}\ \frac{P[z_1/x_1, \ldots, z_n/x_n] \xrightarrow{\alpha} P' \quad (C(x_1, \ldots, x_n) = P, z_i \text{ value of } e_i)}{C(e_1, \ldots, e_n) \xrightarrow{\alpha} P'}$$

Remarks:

- $P[z_1/x_1, \ldots, z_n/x_n]$ denotes the substitution of each free occurrence of x_i by z_i $(1 \leq i \leq n)$.
- Operations on actions ignore values:

  $\overline{a(z)} := \overline{a}(z) \quad \overline{\overline{a}(z)} := a(z) \quad f(a(z)) := f(a)(z) \quad f(\overline{a}(z)) := \overline{f(a)}(z)$ (and $f(\tau) := \tau$).
- The binding restriction ensures that all expressions have a defined value.
- The two-armed conditional if b then P else Q is definable by (if b then P) + (if $\neg b$ then Q).

Semantics of Value-Passing CCS III

Example 12.7

- One-place buffer that outputs non-negative predecessor of stored value:

$$\begin{aligned} B &= in(x).B'(x) \\ B'(x) &= (\text{if } x = 0 \text{ then } \overline{out}(0).B) + (\text{if } x > 0 \text{ then } \overline{out}(x-1).B) \end{aligned}$$

Semantics of Value-Passing CCS III

Example 12.7

- One-place buffer that outputs non-negative predecessor of stored value:

$$
\begin{aligned}
B &= in(x).B'(x) \\
B'(x) &= (\text{if } x = 0 \text{ then } \overline{out}(0).B) + (\text{if } x > 0 \text{ then } \overline{out}(x-1).B)
\end{aligned}
$$

- Input of value "1":

$$
\text{(Call)}\,\frac{\text{(In)}\,\dfrac{}{in(x).B'(x) \xrightarrow{in(1)} B'(1)}}{B \xrightarrow{in(1)} B'(1)}
$$

Semantics of Value-Passing CCS III

Example 12.7

- One-place buffer that outputs non-negative predecessor of stored value:

$$B = in(x).B'(x)$$
$$B'(x) = (\text{if } x = 0 \text{ then } \overline{out}(0).B) + (\text{if } x > 0 \text{ then } \overline{out}(x-1).B)$$

- Input of value "1":

$$\text{(Call)}\,\dfrac{\text{(In)}\,\dfrac{}{in(x).B'(x) \xrightarrow{in(1)} B'(1)}}{B \xrightarrow{in(1)} B'(1)}$$

- Output of predecessor:

$$\text{(Call)}\,\dfrac{\text{(Sum}_2\text{)}\,\dfrac{\text{(If)}\,\dfrac{\text{(Out)}\,\dfrac{}{\overline{out}(1-1).B \xrightarrow{\overline{out}(0)} B}}{\text{if } 1 > 0 \text{ then } \overline{out}(1-1).B \xrightarrow{\overline{out}(0)} B}}{(\text{if } 1 = 0 \text{ then } \overline{out}(0).B) + (\text{if } 1 > 0 \text{ then } \overline{out}(1-1).B) \xrightarrow{\overline{out}(0)} B}}{B'(1) \xrightarrow{\overline{out}(0)} B}$$

Outline of Lecture 12

1. Modelling Mutual Exclusion Algorithms
2. Evaluating the CCS Model
3. Verifying Properties by Model Checking
4. Verifying Mutual Exclusion by Bisimulation Checking
5. Verifying Mutual Exclusion by Testing
6. Syntax of Value-Passing CCS
7. Semantics of Value-Passing CCS
8. Translation of Value-Passing into Pure CCS

Translation of Value-Passing into Pure CCS I

- **To show:** Value-passing process definitions can be represented in pure CCS.
- **Idea:** Each parametrised construct ($a(x)$, $\overline{a}(e)$, $C(e_1, \ldots, e_n)$) corresponds to an indexed family of constructs in pure CCS, one for each possible (combination of) integer value(s).
- Requires extension of pure CCS by infinite choices ("$\sum \ldots$"), restrictions, and process definitions.

Translation of Value-Passing into Pure CCS II

Definition 12.8 (Translation of value-passing into pure CCS)

For each $P \in Prc^+$ without free variables, its translated form $\widehat{P} \in Prc$ is given by

$$\widehat{\mathrm{nil}} := \mathrm{nil} \qquad \widehat{\tau.P} := \tau.\widehat{P}$$

$$\widehat{a(x).P} := \sum_{z \in \mathbb{Z}} a_z.\widehat{P[z/x]} \qquad \widehat{\overline{a}(e).P} := \overline{a_z}.\widehat{P} \quad (z \text{ value of } e)$$

$$\widehat{P_1 + P_2} := \widehat{P_1} + \widehat{P_2} \qquad \widehat{P_1 \parallel P_2} := \widehat{P_1} \parallel \widehat{P_2}$$

$$\widehat{P \setminus L} := \widehat{P} \setminus \{a_z \mid a \in L, z \in \mathbb{Z}\} \qquad \widehat{P[f]} := \widehat{P}[\widehat{f}] \quad (\widehat{f}(a_z) := f(a)_z)$$

$$\widehat{\text{if } b \text{ then } P} := \begin{cases} \widehat{P} & \text{if } b \text{ true} \\ \mathrm{nil} & \text{otherwise} \end{cases} \qquad \widehat{C(e_1, \ldots, e_n)} := C_{z_1,\ldots,z_n} \quad (z_i \text{ value of } e_i)$$

Translation of Value-Passing into Pure CCS II

Definition 12.8 (Translation of value-passing into pure CCS)

For each $P \in Prc^{+}$ without free variables, its translated form $\widehat{P} \in Prc$ is given by

$$\widehat{\mathrm{nil}} := \mathrm{nil} \qquad \widehat{\tau.P} := \tau.\widehat{P}$$

$$\widehat{a(x).P} := \sum_{z \in \mathbb{Z}} a_z.\widehat{P[z/x]} \qquad \widehat{\overline{a}(e).P} := \overline{a_z}.\widehat{P} \quad (z \text{ value of } e)$$

$$\widehat{P_1 + P_2} := \widehat{P_1} + \widehat{P_2} \qquad \widehat{P_1 \parallel P_2} := \widehat{P_1} \parallel \widehat{P_2}$$

$$\widehat{P \setminus L} := \widehat{P} \setminus \{a_z \mid a \in L, z \in \mathbb{Z}\} \qquad \widehat{P[f]} := \widehat{P}[\widehat{f}] \quad (\widehat{f}(a_z) := f(a)_z)$$

$$\widehat{\text{if } b \text{ then } P} := \begin{cases} \widehat{P} & \text{if } b \text{ true} \\ \mathrm{nil} & \text{otherwise} \end{cases} \qquad \widehat{C(e_1, \ldots, e_n)} := C_{z_1,\ldots,z_n} \quad (z_i \text{ value of } e_i)$$

Moreover, each defining equation $C(x_1, \ldots, x_n) = P$ of a process identifier is translated into the indexed collection of process definitions

$$\left(C_{z_1,\ldots,z_n} = \widehat{P[z_1/x_1, \ldots, z_n/x_n]} \mid z_1, \ldots, z_n \in \mathbb{Z} \right)$$

Translation of Value-Passing into Pure CCS III

Example 12.9 (cf. Example 12.7)

$$B = in(x).B'(x)$$
$$B'(x) = (\text{if } x = 0 \text{ then } \overline{out}(0).B) + (\text{if } x > 0 \text{ then } \overline{out}(x-1).B)$$

translates to

$$B = \sum_{z \in \mathbb{Z}} in_z.B'_z$$
$$B'_z = P_z + Q_z \text{ where } P_z := \begin{cases} \overline{out_0}.B & \text{if } z = 0 \\ \text{nil} & \text{otherwise} \end{cases} \text{ and } Q_z := \begin{cases} \overline{out_{z-1}}.B & \text{if } z > 0 \\ \text{nil} & \text{otherwise} \end{cases}$$
(for each $z \in \mathbb{Z}$)

Example 12.9 (cf. Example 12.7)

$$B = in(x).B'(x)$$
$$B'(x) = (\text{if } x = 0 \text{ then } \overline{out}(0).B) + (\text{if } x > 0 \text{ then } \overline{out}(x-1).B)$$

translates to

$$B = \sum_{z \in \mathbb{Z}} in_z.B'_z$$
$$B'_z = P_z + Q_z \text{ where } P_z := \begin{cases} \overline{out_0}.B & \text{if } z = 0 \\ \text{nil} & \text{otherwise} \end{cases} \text{ and } Q_z := \begin{cases} \overline{out_{z-1}}.B & \text{if } z > 0 \\ \text{nil} & \text{otherwise} \end{cases}$$
(for each $z \in \mathbb{Z}$)

Theorem 12.10 (Correctness of translation)

For all $P, P' \in Prc^+$ *and* $\alpha \in Act$,

$$P \xrightarrow{\alpha} P' \iff \widehat{P} \xrightarrow{\widehat{\alpha}} \widehat{P'}$$

where $\widehat{a(z)} := a_z$, $\widehat{\overline{a}(z)} := \overline{a}_z$, *and* $\widehat{\tau} := \tau$.

Translation of Value-Passing into Pure CCS III

Example 12.9 (cf. Example 12.7)

$$B = in(x).B'(x)$$
$$B'(x) = (\text{if } x = 0 \text{ then } \overline{out}(0).B) + (\text{if } x > 0 \text{ then } \overline{out}(x-1).B)$$

translates to

$$B = \sum_{z \in \mathbb{Z}} in_z.B'_z$$
$$B'_z = P_z + Q_z \text{ where } P_z := \begin{cases} \overline{out_0}.B & \text{if } z = 0 \\ nil & \text{otherwise} \end{cases} \text{ and } Q_z := \begin{cases} \overline{out_{z-1}}.B & \text{if } z > 0 \\ nil & \text{otherwise} \end{cases}$$
(for each $z \in \mathbb{Z}$)

Theorem 12.10 (Correctness of translation)

For all $P, P' \in Prc^+$ and $\alpha \in Act$,

$$P \xrightarrow{\alpha} P' \iff \widehat{P} \xrightarrow{\widehat{\alpha}} \widehat{P'}$$

where $\widehat{a(z)} := a_z$, $\widehat{\overline{a}(z)} := \overline{a}_z$, and $\widehat{\tau} := \tau$.

Proof.

by induction on the structure of P (omitted) □