# Concurrency Theory

Winter 2025/26

Lecture 3: Trace Equivalence

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 3

# Syntax of CCS I

## Definition (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 $\tau$.
- Let $Pid$ be a set of process identifiers.
- The set $Prc$ of process expressions is defined by the following syntax:

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

where $\alpha \in Act$, $\emptyset \neq L \subseteq A$, $C \in Pid$, and $f : Act \to Act$ such that $f(\tau) = \tau$ and $f(\overline{a}) = \overline{f(a)}$

# Syntax of CCS II

## Definition (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 $\{C_1, \ldots, C_k\}$).

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

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

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

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

# Semantics of CCS III

## Example (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 (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:

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

# Outline of Lecture 3

# Preliminaries

- When using process algebras like CCS, an important approach is to model both the specification and implementation as CCS processes, say *Spec* and *Impl*.
  - two-place buffer (Example 2.2): sequential “specification” vs. parallel implementation
  - mutual exclusion (later)

# Preliminaries

- When using process algebras like CCS, an important approach is to model both the specification and implementation as CCS processes, say *Spec* and *Impl*.
  - two-place buffer (Example 2.2): sequential “specification” vs. parallel implementation
  - mutual exclusion (later)
- This gives rise to the natural question: when are two CCS processes behaving the same?

# Preliminaries

- When using process algebras like CCS, an important approach is to model both the specification and implementation as CCS processes, say *Spec* and *Impl*.
  - two-place buffer (Example 2.2): sequential “specification” vs. parallel implementation
  - mutual exclusion (later)
- This gives rise to the natural question: when are two CCS processes behaving the same?
- As there are many different interpretations of “behaving the same”, different behavioural equivalences have emerged.

# Behavioural Equivalence

## Implementation

$$\begin{aligned} CM &= coin.\overline{coffee}.CM \\ CS &= \overline{pub}.\overline{coin}.coffee.CS \\ Uni &= (CM \parallel CS) \setminus \{coin, coffee\} \end{aligned}$$

# Behavioural Equivalence

**Implementation**

$$\begin{aligned} CM &= coin.\overline{coffee}.CM \\ CS &= \overline{pub}.\overline{coin}.coffee.CS \\ Uni &= (CM \parallel CS) \setminus \{coin, coffee\} \end{aligned}$$

**Specification**

$$Spec = \overline{pub}.Spec$$

# Behavioural Equivalence

**Implementation**

$$\begin{aligned} CM &= coin.\overline{coffee}.CM \\ CS &= \overline{pub}.\overline{coin}.coffee.CS \\ Uni &= (CM \parallel CS) \setminus \{coin, coffee\} \end{aligned}$$

**Specification**

$$Spec = \overline{pub}.Spec$$

**Question**

Are the specification *Spec* and implementation *Uni* behaviourally equivalent:

$$Spec \stackrel{?}{\equiv} Uni$$

# Equivalence Relations

## Some reasonable required properties

- Reflexivity: $P \equiv P$ for every process $P$
- Symmetry: $P \equiv Q$ if and only if $Q \equiv P$
- Transitivity: $Spec_0 \equiv \ldots \equiv Spec_n \equiv Impl$ implies that $Spec_0 \equiv Impl$

# Equivalence Relations

## Some reasonable required properties

- Reflexivity: $P \equiv P$ for every process $P$
- Symmetry: $P \equiv Q$ if and only if $Q \equiv P$
- Transitivity: $Spec_0 \equiv \ldots \equiv Spec_n \equiv Impl$ implies that $Spec_0 \equiv Impl$

## Definition 3.1 (Equivalence relation)

A binary relation $\equiv \subseteq S \times S$ over a set $S$ is an equivalence if

- it is reflexive: $s \equiv s$ for every $s \in S$,
- it is symmetric: $s \equiv t$ implies $t \equiv s$ for every $s, t \in S$,
- it is transitive: $s \equiv t$ and $t \equiv u$ implies $s \equiv u$ for every $s, t, u \in S$.

# Equivalence Relations

## Some reasonable required properties

- Reflexivity: $P \equiv P$ for every process $P$
- Symmetry: $P \equiv Q$ if and only if $Q \equiv P$
- Transitivity: $Spec_0 \equiv \ldots \equiv Spec_n \equiv Impl$ implies that $Spec_0 \equiv Impl$

## Definition 3.1 (Equivalence relation)

A binary relation $\equiv \subseteq S \times S$ over a set $S$ is an equivalence if

- it is reflexive: $s \equiv s$ for every $s \in S$,
- it is symmetric: $s \equiv t$ implies $t \equiv s$ for every $s, t \in S$,
- it is transitive: $s \equiv t$ and $t \equiv u$ implies $s \equiv u$ for every $s, t, u \in S$.

**Remark:** equivalences induce quotient structures with equivalence classes as elements:

$$S/\!\equiv \;:=\; \{[s]_\equiv \mid s \in S\} \subseteq 2^S \quad \text{where} \quad [s]_\equiv \;:=\; \{s' \in S \mid s' \equiv s\} \subseteq S$$

# Outline of Lecture 3

# Isomorphism: An Example Behavioural Equivalence

## Definition 3.2 (LTS isomorphism)

Two LTSs $T_1 = (S_1, Act_1, \longrightarrow_1)$ and $T_2 = (S_2, Act_2, \longrightarrow_2)$ are isomorphic, denoted $T_1 \equiv_{iso} T_2$, if there exists a bijection $f : S_1 \to S_2$ such that

$$\forall s, \alpha, t. \qquad s \xrightarrow{\alpha}_1 t \quad \text{if and only if} \quad f(s) \xrightarrow{\alpha}_2 f(t).$$

# Isomorphism: An Example Behavioural Equivalence

## Definition 3.2 (LTS isomorphism)

Two LTSs $T_1 = (S_1, Act_1, \longrightarrow_1)$ and $T_2 = (S_2, Act_2, \longrightarrow_2)$ are isomorphic, denoted $T_1 \equiv_{iso} T_2$, if there exists a bijection $f : S_1 \to S_2$ such that

$$\forall s, \alpha, t. \qquad s \xrightarrow{\alpha}_1 t \quad \text{if and only if} \quad f(s) \xrightarrow{\alpha}_2 f(t).$$

It follows immediately that $\equiv_{iso}$ is an equivalence.

# Isomorphism: An Example Behavioural Equivalence

## Definition 3.2 (LTS isomorphism)

Two LTSs $T_1 = (S_1, Act_1, \longrightarrow_1)$ and $T_2 = (S_2, Act_2, \longrightarrow_2)$ are isomorphic, denoted $T_1 \equiv_{iso} T_2$, if there exists a bijection $f : S_1 \to S_2$ such that

$$\forall s, \alpha, t. \qquad s \xrightarrow{\alpha}_1 t \quad \text{if and only if} \quad f(s) \xrightarrow{\alpha}_2 f(t).$$

It follows immediately that $\equiv_{iso}$ is an equivalence.

## Lemma 3.3 (Abelian monoid laws for $+$ and $\parallel$)

*For all CCS processes* $P, Q \in Prc$,

(1) *Commutativity:* $LTS(P + Q) \equiv_{iso} LTS(Q + P)$, $LTS(P \parallel Q) \equiv_{iso} LTS(Q \parallel P)$

(2) *Associativity:* $LTS((P + Q) + R) \equiv_{iso} LTS(P + (Q + R))$,
$LTS((P \parallel Q) \parallel R) \equiv_{iso} LTS(P \parallel (Q \parallel R))$

(3) *Neutral elements:* $LTS(P + \text{nil}) \equiv_{iso} LTS(P \parallel \text{nil}) \equiv_{iso} LTS(P)$

# Isomorphism II

## Assumption

From now on, we consider processes modulo isomorphism, i.e., we do not distinguish CCS processes with isomorphic LTSs.

# Isomorphism II

## Assumption

From now on, we consider processes modulo isomorphism, i.e., we do not distinguish CCS processes with isomorphic LTSs.

## Caveat

Isomorphism is too distinctive. For instance,

$$X = a.X \quad \text{and} \quad Y = a.a.Y$$

are not isomorphic although both can (only) execute infinitely many $a$-actions and should thus be considered equivalent.

# Outline of Lecture 3

# Process Traces I

**Goal:** reduce processes to the sequences of actions they can perform

## Definition 3.4 (Trace language)

For every $P \in Prc$, let

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

be the trace language of $P$ (where $\xrightarrow{w} := \xrightarrow{\alpha_1} \circ \ldots \circ \xrightarrow{\alpha_n}$ for $w = \alpha_1 \ldots \alpha_n$).

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

# Process Traces I

**Goal:** reduce processes to the sequences of actions they can perform

## Definition 3.4 (Trace language)

For every $P \in Prc$, let

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

be the trace language of $P$ (where $\xrightarrow{w} := \xrightarrow{\alpha_1} \circ \ldots \circ \xrightarrow{\alpha_n}$ for $w = \alpha_1 \ldots \alpha_n$).

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

## Example 3.5 (One-place buffer)

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

$\Rightarrow Tr(B) = (in \cdot \overline{out})^* \cdot (in \mid \varepsilon)$

**Remarks:**

- The trace language of $P \in Prc$ is accepted by the LTS of $P$, interpreted as a (finite or infinite) automaton with initial state $P$ and where every state is final.

# Process Traces II

**Remarks:**

- The trace language of $P \in Prc$ 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).

**Remarks:**

- The trace language of $P \in Prc$ 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 isomorphic LTSs: the trace language of a process consists of the (finite) paths in the LTS. Thus:

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

# Process Traces II

**Remarks:**

- The trace language of $P \in Prc$ 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 isomorphic LTSs: the trace language of a process consists of the (finite) paths in the LTS. Thus:

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

- Later we will see: trace equivalence is too coarse, i.e., identifies too many processes $\Rightarrow$ bisimulation

# Outline of Lecture 3

# The Wish List for Behavioural Equivalences

(1) Less distinctive than isomorphism: an equivalence should distinguish less processes than LTS isomorphism does, i.e., $\equiv$ should be coarser than LTS isomorphism:

$$LTS(P) \equiv_{iso} LTS(Q) \Rightarrow P \equiv Q.$$

# The Wish List for Behavioural Equivalences

(1) Less distinctive than isomorphism: an equivalence should distinguish less processes than LTS isomorphism does, i.e., $\equiv$ should be coarser than LTS isomorphism:

$$LTS(P) \equiv_{iso} LTS(Q) \Rightarrow P \equiv Q.$$

(2) More distinctive than trace equivalence: an equivalence should distinguish more processes than trace equivalence does, i.e., $\equiv$ should be finer than trace equivalence:

$$P \equiv Q \Rightarrow Tr(P) = Tr(Q).$$

# The Wish List for Behavioural Equivalences

(1) Less distinctive than isomorphism: an equivalence should distinguish less processes than LTS isomorphism does, i.e., $\equiv$ should be coarser than LTS isomorphism:

$$LTS(P) \equiv_{iso} LTS(Q) \Rightarrow P \equiv Q.$$

(2) More distinctive than trace equivalence: an equivalence should distinguish more processes than trace equivalence does, i.e., $\equiv$ should be finer than trace equivalence:

$$P \equiv Q \Rightarrow Tr(P) = Tr(Q).$$

(3) Congruence property: the equivalence must be substitutive with respect to all CCS operators (in the following).

# The Wish List for Behavioural Equivalences

(1) Less distinctive than isomorphism: an equivalence should distinguish less processes than LTS isomorphism does, i.e., $\equiv$ should be coarser than LTS isomorphism:

$$LTS(P) \equiv_{iso} LTS(Q) \Rightarrow P \equiv Q.$$

(2) More distinctive than trace equivalence: an equivalence should distinguish more processes than trace equivalence does, i.e., $\equiv$ should be finer than trace equivalence:

$$P \equiv Q \Rightarrow Tr(P) = Tr(Q).$$

(3) Congruence property: the equivalence must be substitutive with respect to all CCS operators (in the following).

(4) Deadlock preservation: equivalent processes should have the same deadlock behaviour, i.e., they can either both deadlock, or both cannot (in the following).

# The Wish List for Behavioural Equivalences

(1) Less distinctive than isomorphism: an equivalence should distinguish less processes than LTS isomorphism does, i.e., $\equiv$ should be coarser than LTS isomorphism:

$$LTS(P) \equiv_{iso} LTS(Q) \Rightarrow P \equiv Q.$$

(2) More distinctive than trace equivalence: an equivalence should distinguish more processes than trace equivalence does, i.e., $\equiv$ should be finer than trace equivalence:

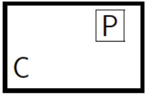
$$P \equiv Q \Rightarrow Tr(P) = Tr(Q).$$

(3) Congruence property: the equivalence must be substitutive with respect to all CCS operators (in the following).

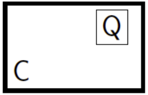
(4) Deadlock preservation: equivalent processes should have the same deadlock behaviour, i.e., they can either both deadlock, or both cannot (in the following).

(5) Optional: the coarsest possible equivalence: there should be no less discriminating equivalence satisfying all these requirements.

# What is a Congruence?

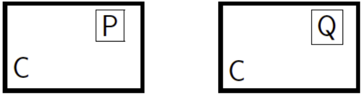



$C(P)$




$C(Q)$

# What is a Congruence?



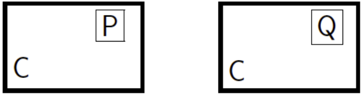

$C(P)$ $C(Q)$

## CCS contexts informally

A CCS context is a CCS process fragment $C(\Box)$ with a "hole" in it, for example:

- $\Box$ (empty context)
- $a.\mathrm{nil} + \Box$
- $(\Box[a \mapsto b] \parallel B) \setminus b$

# What is a Congruence?




$C(P)$ $C(Q)$

## CCS contexts informally

A CCS context is a CCS process fragment $C(\square)$ with a "hole" in it, for example:

- $\square$ (empty context)
- $a.\text{nil} + \square$
- $(\square[a \mapsto b] \parallel B) \setminus b$

## CCS congruences informally

Equivalence relation $\equiv$ is a CCS congruence whenever $P \equiv Q$ implies $C(P) \equiv C(Q)$ for every CCS context $C$.

# The Importance of Congruences

## CCS congruences informally

Equivalence relation $\equiv$ is a CCS congruence whenever $P \equiv Q$ implies $C(P) \equiv C(Q)$ for every CCS context $C$.

# The Importance of Congruences

## CCS congruences informally

Equivalence relation $\equiv$ is a CCS congruence whenever $P \equiv Q$ implies $C(P) \equiv C(Q)$ for every CCS context $C$.

## Example 3.6 (Congruence)

Let $a \equiv b$ for $a, b \in \mathbb{Z}$ whenever $a \bmod k = b \bmod k$, for some $k \in \mathbb{N}_+$.
Equivalence relation $\equiv$ is a congruence for addition and multiplication.

# The Importance of Congruences

## CCS congruences informally

Equivalence relation $\equiv$ is a CCS congruence whenever $P \equiv Q$ implies $C(P) \equiv C(Q)$ for every CCS context $C$.

## Example 3.6 (Congruence)

Let $a \equiv b$ for $a, b \in \mathbb{Z}$ whenever $a \bmod k = b \bmod k$, for some $k \in \mathbb{N}_+$.
Equivalence relation $\equiv$ is a congruence for addition and multiplication.

Important motivations for requiring $\equiv$ to be a congruence on processes:

(1) Model-based development through refinement:
Replacing (part of) an abstract model *Spec* by a more detailed model *Impl*.

(2) Optimisation:
Replacing (part of) an implementation *Impl* by a more efficient implementation *Impl'*.

# CCS Congruences Formally

## Definition 3.7 (CCS congruence)

An equivalence relation $\equiv \subseteq Prc \times Prc$ is a CCS congruence if it is preserved by all CCS constructs, i.e., if $P, Q \in Prc$ with $P \equiv Q$ then:

$$
\begin{array}{rl}
\alpha.P \equiv \alpha.Q & \text{for every } \alpha \in Act \\
P + R \equiv Q + R & \text{for every } R \in Prc \\
P \parallel R \equiv Q \parallel R & \text{for every } R \in Prc \\
P \setminus L \equiv Q \setminus L & \text{for every } L \subseteq A \\
P[f] \equiv Q[f] & \text{for every } f : A \to A
\end{array}
$$

# CCS Congruences Formally

## Definition 3.7 (CCS congruence)

An equivalence relation $\equiv \subseteq Prc \times Prc$ is a CCS congruence if it is preserved by all CCS constructs, i.e., if $P, Q \in Prc$ with $P \equiv Q$ then:

$$
\begin{aligned}
\alpha.P \equiv \alpha.Q & \quad \text{for every } \alpha \in Act \\
P + R \equiv Q + R & \quad \text{for every } R \in Prc \\
P \parallel R \equiv Q \parallel R & \quad \text{for every } R \in Prc \\
P \setminus L \equiv Q \setminus L & \quad \text{for every } L \subseteq A \\
P[f] \equiv Q[f] & \quad \text{for every } f : A \to A
\end{aligned}
$$

Thus, a CCS congruence is substitutive for all possible CCS contexts.

# Deadlocks

## Definition 3.8 (Deadlock)

Let $P, Q \in Prc$ and $w \in Act^*$ such that

$$P \xrightarrow{w} Q \quad \text{and} \quad Q \not\longrightarrow .$$

Then $Q$ is called a *$w$-deadlock* of $P$.

# Deadlocks

## Definition 3.8 (Deadlock)

Let $P, Q \in Prc$ and $w \in Act^*$ such that

$$P \xrightarrow{w} Q \quad \text{and} \quad Q \not\longrightarrow .$$

Then $Q$ is called a $w$-deadlock of $P$.

## Example 3.9

$P = a.b.\text{nil} + a.\text{nil}$ has an $a$-deadlock, whereas $Q = a.b.\text{nil}$ has not.
Such properties are important as it can be crucial that a certain action is eventually enabled.

# Deadlocks

## Definition 3.8 (Deadlock)

Let $P, Q \in Prc$ and $w \in Act^*$ such that

$$P \xrightarrow{w} Q \quad \text{and} \quad Q \not\longrightarrow .$$

Then $Q$ is called a $w$-deadlock of $P$.

## Example 3.9

$P = a.b.\text{nil} + a.\text{nil}$ has an $a$-deadlock, whereas $Q = a.b.\text{nil}$ has not.
Such properties are important as it can be crucial that a certain action is eventually enabled.

## Definition 3.10 (Deadlock sensitivity)

Relation $\equiv \subseteq Prc \times Prc$ is deadlock sensitive whenever:

$$P \equiv Q \quad \text{implies} \quad (\forall w \in Act^* : P \text{ has a } w\text{-deadlock iff } Q \text{ has a } w\text{-deadlock}) .$$

# Outline of Lecture 3

# Checking Trace Equivalence

## Traces by automata

For finite-state $P \in Prc$, the trace language $Tr(P)$ of process $P$ is accepted by the (non-deterministic) finite automaton obtained from the LTS of $P$ with initial state $P$ and making all states accepting (final).

# Checking Trace Equivalence

## Traces by automata

For finite-state $P \in Prc$, the trace language $Tr(P)$ of process $P$ is accepted by the (non-deterministic) finite automaton obtained from the LTS of $P$ with initial state $P$ and making all states accepting (final).

## Theorem 3.11

*Checking trace equivalence of two finite processes is PSPACE-complete.*

# Checking Trace Equivalence

### Traces by automata

For finite-state $P \in Prc$, the trace language $Tr(P)$ of process $P$ is accepted by the (non-deterministic) finite automaton obtained from the LTS of $P$ with initial state $P$ and making all states accepting (final).

### Theorem 3.11

*Checking trace equivalence of two finite processes is PSPACE-complete.*

### Proof.

Checking whether $Tr(P) = Tr(Q)$, for finite-state $P$ and $Q$, boils down to deciding whether their non-deterministic automata accept the same language. As this problem in automata theory is PSPACE-complete, it follows that checking $Tr(P) = Tr(Q)$ is PSPACE-complete. □

# Trace Equivalence is a Congruence

**Theorem 3.12**

*Trace equivalence is a CCS congruence.*

# Trace Equivalence is a Congruence

**Theorem 3.12**

*Trace equivalence is a CCS congruence.*

**Proof.**

By structural induction over the syntax of CCS processes.

# Trace Equivalence is a Congruence

**Theorem 3.12**

*Trace equivalence is a CCS congruence.*

**Proof.**

By structural induction over the syntax of CCS processes.

For $+$ the proof proceeds as follows:

# Trace Equivalence is a Congruence

## Theorem 3.12

*Trace equivalence is a CCS congruence.*

## Proof.

By structural induction over the syntax of CCS processes.

For $+$ the proof proceeds as follows:

- Let $P, Q \in \mathit{Prc}$ with $\mathit{Tr}(P) = \mathit{Tr}(Q)$.

# Trace Equivalence is a Congruence

## Theorem 3.12

*Trace equivalence is a CCS congruence.*

## Proof.

By structural induction over the syntax of CCS processes.

For $+$ the proof proceeds as follows:

- Let $P, Q \in Prc$ with $Tr(P) = Tr(Q)$.
- Then for $R \in Prc$ it holds:

$$Tr(P + R) = Tr(P) \cup Tr(R) = Tr(Q) \cup Tr(R) = Tr(Q + R).$$

# Trace Equivalence is a Congruence

## Theorem 3.12

*Trace equivalence is a CCS congruence.*

## Proof.

By structural induction over the syntax of CCS processes.
For $+$ the proof proceeds as follows:

- Let $P, Q \in Prc$ with $Tr(P) = Tr(Q)$.
- Then for $R \in Prc$ it holds:

$$Tr(P + R) = Tr(P) \cup Tr(R) = Tr(Q) \cup Tr(R) = Tr(Q + R).$$

- Thus, $P + R$ and $Q + R$ are trace equivalent.

# Trace Equivalence is a Congruence

## Theorem 3.12

*Trace equivalence is a CCS congruence.*

## Proof.

By structural induction over the syntax of CCS processes.
For $+$ the proof proceeds as follows:

- Let $P, Q \in Prc$ with $Tr(P) = Tr(Q)$.
- Then for $R \in Prc$ it holds:

$$Tr(P + R) = Tr(P) \cup Tr(R) = Tr(Q) \cup Tr(R) = Tr(Q + R).$$

- Thus, $P + R$ and $Q + R$ are trace equivalent.

For the other CCS constructs, the proof goes along similar lines. □

## Example 3.13 (Coffee/tea machines)

Consider the coffee/tea machine $CTM$ and its variant $CTM'$:

$$\begin{aligned} CTM &= coin.\left(\overline{coffee}.CTM + \overline{tea}.CTM\right) \\ CTM' &= coin.\overline{coffee}.CTM' + coin.\overline{tea}.CTM'. \end{aligned}$$

# Trace Equivalence is not Deadlock Sensitive

## Example 3.13 (Coffee/tea machines)

Consider the coffee/tea machine $CTM$ and its variant $CTM'$:

$$\begin{aligned} CTM &= coin.\left(\overline{coffee}.CTM + \overline{tea}.CTM\right) \\ CTM' &= coin.\overline{coffee}.CTM' + coin.\overline{tea}.CTM'. \end{aligned}$$

Note the difference between the two processes. Nevertheless:

$$Tr(CTM) = Tr(CTM').$$

# Trace Equivalence is not Deadlock Sensitive

## Example 3.13 (Coffee/tea machines)

Consider the coffee/tea machine $CTM$ and its variant $CTM'$:

$$\begin{aligned} CTM &= coin.\left(\overline{coffee}.CTM + \overline{tea}.CTM\right) \\ CTM' &= coin.\overline{coffee}.CTM' + coin.\overline{tea}.CTM'. \end{aligned}$$

Note the difference between the two processes. Nevertheless:

$$Tr(CTM) = Tr(CTM').$$

Are we satisfied?

## Example 3.13 (Coffee/tea machines)

Consider the coffee/tea machine $CTM$ and its variant $CTM'$:

$$\begin{aligned} CTM &= coin.\left(\overline{coffee}.CTM + \overline{tea}.CTM\right) \\ CTM' &= coin.\overline{coffee}.CTM' + coin.\overline{tea}.CTM'. \end{aligned}$$

Note the difference between the two processes. Nevertheless:

$$Tr(CTM) = Tr(CTM').$$

Are we satisfied?

No, as $CTM$ and $CTM'$ differ in the context:

$$C(\square) = (\underbrace{\square}_{\text{hole}} \parallel CA) \setminus \{coin, coffee, tea\} \quad \text{with} \quad CA = \overline{coin}.coffee.CA.$$

# Trace Equivalence is not Deadlock Sensitive

## Example 3.13 (Coffee/tea machines)

Consider the coffee/tea machine $CTM$ and its variant $CTM'$:

$$\begin{array}{rcl} CTM & = & coin.\left(\overline{coffee}.CTM + \overline{tea}.CTM\right) \\ CTM' & = & coin.\overline{coffee}.CTM' + coin.\overline{tea}.CTM'. \end{array}$$

Note the difference between the two processes. Nevertheless:

$$Tr(CTM) = Tr(CTM').$$

Are we satisfied?

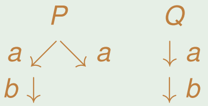
No, as $CTM$ and $CTM'$ differ in the context:

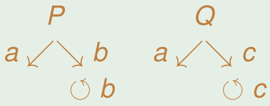
$$C(\square) = (\underbrace{\square}_{\text{hole}} \parallel CA) \setminus \{coin, coffee, tea\} \quad \text{with} \quad CA = \overline{coin}.coffee.CA.$$

Why? $C(CTM')$ may yield a deadlock, but $C(CTM)$ does not.

## Example 3.14 (Traces and deadlocks)

Traces and deadlocks are independent in the following sense:




same traces
different deadlocks

different traces
same deadlocks

# Traces and Deadlocks

## Example 3.14 (Traces and deadlocks)

Traces and deadlocks are independent in the following sense:


| | |
|---|---|
| same traces<br>different deadlocks | different traces<br>same deadlocks |

**But:** processes with finite trace sets and identical deadlocks are trace equivalent (since every trace is a prefix of some deadlock).

# Outline of Lecture 3

An attempt to fix the deadlock sensitivity flaw:

### Definition 3.15 (Completed traces)

A completed trace of $P \in Prc$ is a sequence $w \in Act^*$ such that:

$$P \xrightarrow{w} Q \quad \text{and} \quad Q \not\longrightarrow$$

for some $Q \in Prc$.

# Completed Trace Equivalence

An attempt to fix the deadlock sensitivity flaw:

**Definition 3.15 (Completed traces)**

A completed trace of $P \in Prc$ is a sequence $w \in Act^*$ such that:

$$P \xrightarrow{w} Q \quad \text{and} \quad Q \not\longrightarrow$$

for some $Q \in Prc$.

The completed traces of process $P$ may be seen as capturing its deadlock behaviour, as they are precisely the action sequences that can lead to a process from which no transition is possible (i.e., a deadlock).

# Completed Trace Equivalence

An attempt to fix the deadlock sensitivity flaw:

**Definition 3.15 (Completed traces)**

A completed trace of $P \in Prc$ is a sequence $w \in Act^*$ such that:

$$P \xrightarrow{w} Q \quad \text{and} \quad Q \not\longrightarrow$$

for some $Q \in Prc$.

The completed traces of process $P$ may be seen as capturing its deadlock behaviour, as they are precisely the action sequences that can lead to a process from which no transition is possible (i.e., a deadlock).

**Example 3.16**

- $P = a.b.\text{nil} + a.c.\text{nil}$ and $Q = a.(b.\text{nil} + c.\text{nil})$ have the same completed traces: $\{ab, ac\}$.
- However this does not apply to $P \setminus b$: $\{a, ac\}$ and $Q \setminus b$: $\{ac\}$.
- Thus, completed trace equivalence is *not* a CCS congruence.

# Outline of Lecture 3

# Summary

(1) Behavioural equivalences should be
   (a) less distinctive than isomorphism
   (b) more distinctive than trace equivalence
   (c) CCS congruences
   (d) deadlock sensitive

# Summary

(1) Behavioural equivalences should be
   (a) less distinctive than isomorphism
   (b) more distinctive than trace equivalence
   (c) CCS congruences
   (d) deadlock sensitive
(2) Trace equivalence
   (a) equates processes that have the same traces, i.e., action sequences
   (b) is implied by LTS isomorphism
   (c) is a CCS congruence
   (d) is *not* deadlock sensitive
   (e) checking trace equivalence is PSPACE-complete