

Modeling Concurrent and Probabilistic Systems

Lecture 3: Equivalence of CCS Processes

Joost-Pieter Katoen Thomas Noll

Software Modeling and Verification Group
RWTH Aachen University
noll@cs.rwth-aachen.de

<http://www-i2.informatik.rwth-aachen.de/i2/mcps07/>

Winter Semester 2007/08

1 Repetition: Syntax and Semantics of CCS

2 Recursive Processes

3 Equivalence of CCS Processes

4 Trace Equivalence

5 Deadlocks

Definition (Syntax of CCS)

- Let N be a set of **(action) names**.
- $\overline{N} := \{\bar{a} \mid a \in N\}$ denotes the set of **co-names**.
- $Act := N \cup \overline{N} \cup \{\tau\}$ 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:
$$P ::= \begin{array}{ll} \text{nil} & \text{(inaction)} \\ | & \alpha.P & \text{(prefixing)} \\ | & P_1 + P_2 & \text{(choice)} \\ | & P_1 \parallel P_2 & \text{(parallel composition)} \\ | & \text{new } a \, P & \text{(restriction)} \\ | & A(a_1, \dots, a_n) & \text{(process call)} \end{array}$$
where $\alpha \in Act$, $a, a_i \in N$, and $A \in Pid$.

Definition (continued)

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

$$(A_i(a_{i1}, \dots, a_{in_i}) = P_i \mid 1 \leq i \leq k)$$

where $k \geq 1$, $A_i \in Pid$ (pairwise different), $a_{ij} \in N$, and $P_i \in Prc$ (with process identifiers from $\{A_1, \dots, A_k\}$).

Repetition: Labeled Transition Systems

Goal: represent behavior of system by (infinite) graph

- nodes = system states
- edges = transitions between states

Definition (Labeled transition system)

A **(*Act*-)labeled 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** $\longrightarrow \subseteq S \times Act \times S$

If $(s, \alpha, s') \in \longrightarrow$ 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
- (finite) LTSs correspond to (finite) **automata** without final states

Repetition: Semantics of CCS

Definition (Semantics of CCS)

A process definition $(A_i(a_{i1}, \dots, a_{in_i}) = P_i \mid 1 \leq i \leq k)$ determines the LTS $(Prc, Act, \longrightarrow)$ whose transitions can be inferred from the following rules ($P, P', Q, Q' \in Prc$, $\alpha \in Act$, $\lambda \in N \cup \overline{N}$, $a \in N$):

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \text{(Act)}$$

$$\frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\bar{\lambda}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \text{(Com)}$$

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

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

$$\frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \text{(Par}_1\text{)}$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'} \text{(Par}_2\text{)}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin \{a, \bar{a}\}}{\text{new } a \ P \xrightarrow{\alpha} \text{new } a \ P'} \text{(New)}$$

$$\frac{A(\vec{a}) = P \quad P[\vec{a} \mapsto \vec{b}] \xrightarrow{\alpha} P'}{A(\vec{b}) \xrightarrow{\alpha} P'} \text{(Call)}$$

(Here $P[\vec{a} \mapsto \vec{b}]$ denotes the replacement of every a_i by b_i in P .)

1 Repetition: Syntax and Semantics of CCS

2 Recursive Processes

3 Equivalence of CCS Processes

4 Trace Equivalence

5 Deadlocks

Recursive Processes

Here: recursive processes defined using **equations** such as

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

(simultaneous recursion)

Alternative: explicit fixpoint operator

- syntax: $P ::= \text{nil} \mid \dots \mid \text{fix } A \ P \in \text{Pre}$ (where $A \in \text{Pid}$)
- semantics:
$$\frac{P[A \mapsto P] \xrightarrow{\alpha} P'}{\text{fix } A \ P \xrightarrow{\alpha} \text{fix } A \ P'} \text{ (Fix)}$$
$$\frac{}{\overline{in.\overline{out}.in.\overline{out}.B} \xrightarrow{in} \overline{out.in.\overline{out}.B} \quad \text{ (Act)}} \text{ (Fix)}$$
- example:
$$\frac{\overline{in.\overline{out}.in.\overline{out}.B} \xrightarrow{in} \overline{out.in.\overline{out}.B} \quad \text{ (Fix)}}{\text{fix } B \ \overline{in.\overline{out}.B} \xrightarrow{in} \text{fix } B \ \overline{out.in.\overline{out}.B}}$$

(nested scalar recursion)

Advantage: only process term level required (no equations)
 \implies simplification of theory

Disadvantage: bad readability of process definitions

Recursive Processes

Here: recursive processes defined using **equations** such as

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

(simultaneous recursion)

Alternative: explicit **fixpoint operator**

- syntax: $P ::= \text{nil} \mid \dots \mid \text{fix } A \ P \in Prc$ (where $A \in Pid$)
- semantics:
$$\frac{P[A \mapsto P] \xrightarrow{\alpha} P'}{\text{fix } A \ P \xrightarrow{\alpha} \text{fix } A \ P'} \text{ (Fix)}$$
- example:
$$\frac{\overline{in.\overline{out}.in.\overline{out}.B} \xrightarrow{in} \overline{out.in.\overline{out}.B}}{\text{fix } B \ \overline{in.\overline{out}.B} \xrightarrow{in} \text{fix } B \ \overline{out.in.\overline{out}.B}} \text{ (Act)}$$
 (Fix)

(nested scalar recursion)

Advantage: only process term level required (no equations)
⇒ simplification of theory

Disadvantage: bad readability of process definitions

Recursive Processes

Here: recursive processes defined using **equations** such as

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

(simultaneous recursion)

Alternative: explicit **fixpoint operator**

- syntax: $P ::= \text{nil} \mid \dots \mid \text{fix } A \ P \in Prc$ (where $A \in Pid$)
- semantics:
$$\frac{P[A \mapsto P] \xrightarrow{\alpha} P'}{\text{fix } A \ P \xrightarrow{\alpha} \text{fix } A \ P'} \text{ (Fix)}$$
- example:
$$\frac{\overline{in.\overline{out}.in.\overline{out}.B} \xrightarrow{in} \overline{out.in.\overline{out}.B}}{\text{fix } B \ \overline{in.\overline{out}.B} \xrightarrow{in} \text{fix } B \ \overline{out.in.\overline{out}.B}} \text{ (Act)}$$
 (Fix)

(nested scalar recursion)

Advantage: only process term level required (no equations)
 \implies simplification of theory

Disadvantage: bad readability of process definitions

- 1 Repetition: Syntax and Semantics of CCS
- 2 Recursive Processes
- 3 Equivalence of CCS Processes
- 4 Trace Equivalence
- 5 Deadlocks

Goal: identify process expressions which have the same “meaning” but differ in their syntax

Definition 3.1 (Equivalence relation)

Let $\cong \subseteq S \times S$ be a binary relation over some set S . Then \cong is called an **equivalence relation** if it is

- **reflexive**, i.e., $s \cong s$ for every $s \in S$,
- **symmetric**, i.e., $s \cong t$ implies $t \cong s$ for every $s, t \in S$, and
- **transitive**, i.e., $s \cong t$ and $t \cong u$ implies $s \cong u$ for every $s, t, u \in S$.

- **Generally:** two syntactic objects are equivalent if they have the same “meaning”
- **Here:** two processes are equivalent if they have the same “behavior” (i.e., communication potential)
- Communication potential described by LTS
- **Idea:** choose
meaning of a process $P := LTS(P)$
- **But:** yields too many distinctions:

Example 3.2

$$X(a) = a.X(a) \quad Y(a) = a.a.Y(a)$$

LTS:



although both processes can (only) execute infinitely many a -actions,
and should be considered equivalent therefore

- **Generally:** two syntactic objects are equivalent if they have the same “meaning”
- **Here:** two processes are equivalent if they have the same “behavior” (i.e., communication potential)
- Communication potential described by LTS
- Idea: choose meaning of a process $P := LTS(P)$
- But: yields too many distinctions:

Example 3.2

$$X(a) = a.X(a) \quad Y(a) = a.a.Y(a)$$



although both processes can (only) execute infinitely many a -actions, and should be considered equivalent therefore

- **Generally:** two syntactic objects are equivalent if they have the same “meaning”
- **Here:** two processes are equivalent if they have the same “behavior” (i.e., communication potential)
- Communication potential described by LTS
- **Idea:** choose
meaning of a process $P := LTS(P)$
- **But:** yields too many distinctions:

Example 3.2

$$X(a) = a.X(a) \quad Y(a) = a.a.Y(a)$$

LTS:



although both processes can (only) execute infinitely many a -actions,
and should be considered equivalent therefore

- **Generally:** two syntactic objects are equivalent if they have the same “meaning”
- **Here:** two processes are equivalent if they have the same “behavior” (i.e., communication potential)
- Communication potential described by LTS
- **Idea:** choose
meaning of a process $P := LTS(P)$
- **But:** yields too many distinctions:

Example 3.2

$$X(a) = a.X(a) \quad Y(a) = a.a.Y(a)$$



although both processes can (only) execute infinitely many a -actions, and should be considered equivalent therefore

Desired Properties of Equivalence

Wanted: a “feasible” (i.e., efficiently decidable) semantic equivalence between CCS processes which

- ① identifies processes whose **LTSs coincide**,
- ② implies trace equivalence, i.e., considers two processes equivalent only if both can execute the same actions sequences (formal definition later), and
- ③ is a **congruence**, i.e., allows to replace a subprocess by an equivalent counterpart without changing the overall semantics of the system (formal definition later).

Formally: we are looking for a congruence relation $\cong \subseteq Prc \times Prc$ such that

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

Desired Properties of Equivalence

Wanted: a “feasible” (i.e., efficiently decidable) semantic equivalence between CCS processes which

- ① identifies processes whose **LTSs coincide**,
- ② implies **trace equivalence**, i.e., considers two processes equivalent only if both can execute the same actions sequences (formal definition later), and
- ③ is a **congruence**, i.e., allows to replace a subprocess by an equivalent counterpart without changing the overall semantics of the system (formal definition later).

Formally: we are looking for a congruence relation $\cong \subseteq Prc \times Prc$ such that

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

Desired Properties of Equivalence

Wanted: a “feasible” (i.e., efficiently decidable) semantic equivalence between CCS processes which

- ① identifies processes whose **LTSs coincide**,
- ② **implies trace equivalence**, i.e., considers two processes equivalent only if both can execute the same actions sequences (formal definition later), and
- ③ is a **congruence**, i.e., allows to replace a subprocess by an equivalent counterpart without changing the overall semantics of the system (formal definition later).

Formally: we are looking for a congruence relation $\cong \subseteq Prc \times Prc$ such that

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

Desired Properties of Equivalence

Wanted: a “feasible” (i.e., efficiently decidable) semantic equivalence between CCS processes which

- ① identifies processes whose **LTSs coincide**,
- ② **implies trace equivalence**, i.e., considers two processes equivalent only if both can execute the same actions sequences (formal definition later), and
- ③ is a **congruence**, i.e., allows to replace a subprocess by an equivalent counterpart without changing the overall semantics of the system (formal definition later).

Formally: we are looking for a congruence relation $\cong \subseteq Prc \times Prc$ such that

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

Goal: replacing a subcomponent of a system by an equivalent process should yield an equivalent systems
⇒ modular system development

Definition 3.3 (CCS congruence)

An equivalence relation $\cong \subseteq \text{Pre} \times \text{Pre}$ is said to be a **CCS congruence** if it is preserved by the CCS constructs; that is, if $P, Q, R \in \text{Pre}$ such that $P \cong Q$ then

$$\begin{aligned}\alpha.P &\cong \alpha.Q \\ P + R &\cong Q + R \\ R + P &\cong R + Q \\ P \parallel R &\cong Q \parallel R \\ R \parallel P &\cong R \parallel Q \\ \text{new } a P &\cong \text{new } a Q\end{aligned}$$

for every $\alpha \in \text{Act}$ and $a \in N$.

Goal: replacing a subcomponent of a system by an equivalent process should yield an equivalent systems
⇒ modular system development

Definition 3.3 (CCS congruence)

An equivalence relation $\cong \subseteq Prc \times Prc$ is said to be a **CCS congruence** if it is preserved by the CCS constructs; that is, if $P, Q, R \in Prc$ such that $P \cong Q$ then

$$\begin{aligned}\alpha.P &\cong \alpha.Q \\ P + R &\cong Q + R \\ R + P &\cong R + Q \\ P \parallel R &\cong Q \parallel R \\ R \parallel P &\cong R \parallel Q \\ \text{new } a \, P &\cong \text{new } a \, Q\end{aligned}$$

for every $\alpha \in Act$ and $a \in N$.

- 1 Repetition: Syntax and Semantics of CCS
- 2 Recursive Processes
- 3 Equivalence of CCS Processes
- 4 Trace Equivalence
- 5 Deadlocks

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 .

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

Example 3.5 (One-place buffer)

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

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

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 .

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

Example 3.5 (One-place buffer)

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

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

Remarks:

- The trace language of $P \in Prc$ is accepted by the LTS of P , interpreted as an automaton where **every state is final**.
- Trace equivalence is obviously an **equivalence relation** (i.e., reflexive, symmetric, and transitive).
- Trace equivalence possesses the postulated properties of a process equivalence:
 - ➊ it identifies processes with **identical LTSs**: the trace language of a process consists of the (finite) paths in the LTS. Hence processes with identical LTSs are trace equivalent.
 - ➋ it **implies trace equivalence**: trivial
 - ➌ it is a **congruence**:

Remarks:

- The trace language of $P \in Prc$ is accepted by the LTS of P , interpreted as an automaton where **every state is final**.
- Trace equivalence is obviously an **equivalence relation** (i.e., reflexive, symmetric, and transitive).
- Trace equivalence possesses the postulated properties of a process equivalence:
 - ➊ it identifies processes with **identical LTSs**: the trace language of a process consists of the (finite) paths in the LTS. Hence processes with identical LTSs are trace equivalent.
 - ➋ it **implies trace equivalence**: trivial
 - ➌ it is a **congruence**:

Remarks:

- The trace language of $P \in Prc$ is accepted by the LTS of P , interpreted as an automaton where **every state is final**.
- Trace equivalence is obviously an **equivalence relation** (i.e., reflexive, symmetric, and transitive).
- Trace equivalence possesses the postulated properties of a process equivalence:
 - ➊ it identifies processes with **identical LTSs**: the trace language of a process consists of the (finite) paths in the LTS. Hence processes with identical LTSs are trace equivalent.
 - ➋ it **implies trace equivalence**: trivial
 - ➌ it is a **congruence**:

CCS Congruences [repetition]

Goal: replacing a subcomponent of a system by an equivalent process should yield an equivalent systems
⇒ modular system development

Definition (CCS congruence)

An equivalence relation $\cong \subseteq Prc \times Prc$ is said to be a **CCS congruence** if it is preserved by the CCS constructs; that is, if $P \cong Q$ then

$$\begin{aligned}\alpha.P &\cong \alpha.Q \\ P + R &\cong Q + R \\ R + P &\cong R + Q \\ P \parallel R &\cong Q \parallel R \\ R \parallel P &\cong R \parallel Q \\ \text{new } a \ P &\cong \text{new } a \ Q\end{aligned}$$

for every $\alpha \in Act$, $R \in Prc$, and $a \in N$.

Theorem 3.6

Trace equivalence is a congruence.

Proof.

(only for $+$; remaining operators analogously)

Clearly we have:

$$Tr(P_1 + P_2) = Tr(P_1) \cup Tr(P_2)$$

Now let $P, Q, R \in Prc$ with $Tr(P) = Tr(Q)$. Then:

$$\begin{array}{ll} Tr(P + R) & Tr(R + P) \\ = Tr(P) \cup Tr(R) & = Tr(R) \cup Tr(P) \\ = Tr(Q) \cup Tr(R) & = Tr(R) \cup Tr(Q) \\ = Tr(Q + R) & = Tr(R + Q) \\ \implies P + R, Q + R \text{ trace equiv.} & \implies R + P, R + Q \text{ trace equiv.} \end{array}$$

□

Trace Equivalence III

Theorem 3.6

Trace equivalence is a congruence.

Proof.

(only for $+$; remaining operators analogously)

Clearly we have:

$$Tr(P_1 + P_2) = Tr(P_1) \cup Tr(P_2)$$

Now let $P, Q, R \in Prc$ with $Tr(P) = Tr(Q)$. Then:

$$\begin{array}{ll} Tr(P + R) & Tr(R + P) \\ = Tr(P) \cup Tr(R) & = Tr(R) \cup Tr(P) \\ = Tr(Q) \cup Tr(R) & = Tr(R) \cup Tr(Q) \\ = Tr(Q + R) & = Tr(R + Q) \\ \Rightarrow P + R, Q + R \text{ trace equiv.} & \Rightarrow R + P, R + Q \text{ trace equiv.} \end{array}$$



Theorem 3.6

Trace equivalence is a congruence.

Proof.

(only for $+$; remaining operators analogously)

Clearly we have:

$$Tr(P_1 + P_2) = Tr(P_1) \cup Tr(P_2)$$

Now let $P, Q, R \in Prc$ with $Tr(P) = Tr(Q)$. Then:

$$\begin{array}{ll} Tr(P + R) & Tr(R + P) \\ = Tr(P) \cup Tr(R) & = Tr(R) \cup Tr(P) \\ = Tr(Q) \cup Tr(R) & = Tr(R) \cup Tr(Q) \\ = Tr(Q + R) & = Tr(R + Q) \\ \implies P + R, Q + R \text{ trace equiv.} & \implies R + P, R + Q \text{ trace equiv.} \end{array}$$



- We have found a process equivalence with the three required properties.
- Are we satisfied? No!

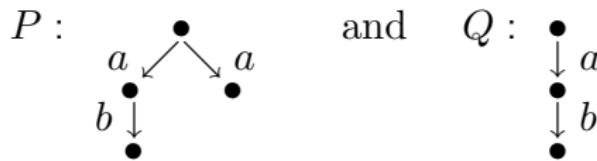


are trace equivalent ($Tr(P) = Tr(Q) = \{\varepsilon, a, ab\}$)

- But P and Q are **distinguishable**:
 - both can execute ab
 - but P can deny b
 - while Q always has to offer b after a

\implies take into account such **deadlock properties**

- We have found a process equivalence with the three required properties.
- Are we satisfied? No!



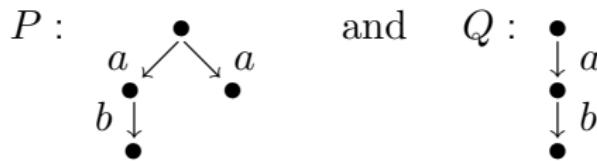
are trace equivalent ($Tr(P) = Tr(Q) = \{\varepsilon, a, ab\}$)

- But P and Q are distinguishable:

- both can execute ab
- but P can deny b
- while Q always has to offer b after a

\implies take into account such **deadlock properties**

- We have found a process equivalence with the three required properties.
- Are we satisfied? No!



are trace equivalent ($Tr(P) = Tr(Q) = \{\varepsilon, a, ab\}$)

- But P and Q are **distinguishable**:
 - both can execute ab
 - but P can deny b
 - while Q always has to offer b after a

\implies take into account such **deadlock properties**

- 1 Repetition: Syntax and Semantics of CCS
- 2 Recursive Processes
- 3 Equivalence of CCS Processes
- 4 Trace Equivalence
- 5 Deadlocks

Definition 3.7 (Deadlock)

Let $P, Q \in Prc$ and $w \in Act^*$ such that $P \xrightarrow{w}^* Q$ and $Q \not\rightarrow$. Then Q is called a **w -deadlock** of P .

- Thus $P := a.b.nil + a.nil$ has an a -deadlock, in contrast to $Q := a.b.nil$.
- Such properties are important since it can be crucial that a certain communication is eventually possible.
- We therefore extend our set of postulates: our semantic equivalence \cong should
 - ➊ identify processes with identical LTSs;
 - ➋ imply trace equivalence;
 - ➌ be a congruence; and
 - ➍ be **deadlock sensitive**, i.e., if $P \cong Q$ and if P has a w -deadlock, then Q has a w -deadlock (and vice versa, by equivalence).

Definition 3.7 (Deadlock)

Let $P, Q \in Prc$ and $w \in Act^*$ such that $P \xrightarrow{w}^* Q$ and $Q \not\rightarrow$. Then Q is called a **w -deadlock** of P .

- Thus $P := a.b.\text{nil} + a.\text{nil}$ has an a -deadlock, in contrast to $Q := a.b.\text{nil}$.
- Such properties are important since it can be crucial that a certain communication is eventually possible.
- We therefore extend our set of postulates: our semantic equivalence \cong should
 - ➊ identify processes with identical LTSs;
 - ➋ imply trace equivalence;
 - ➌ be a congruence; and
 - ➍ be **deadlock sensitive**, i.e., if $P \cong Q$ and if P has a w -deadlock, then Q has a w -deadlock (and vice versa, by equivalence).

Definition 3.7 (Deadlock)

Let $P, Q \in Prc$ and $w \in Act^*$ such that $P \xrightarrow{w}^* Q$ and $Q \not\rightarrow$. Then Q is called a **w -deadlock** of P .

- Thus $P := a.b.\text{nil} + a.\text{nil}$ has an a -deadlock, in contrast to $Q := a.b.\text{nil}$.
- Such properties are important since it can be crucial that a certain communication is eventually possible.
- We therefore extend our set of postulates: our semantic equivalence \cong should
 - ➊ identify processes with identical LTSs;
 - ➋ imply trace equivalence;
 - ➌ be a congruence; and
 - ➍ be **deadlock sensitive**, i.e., if $P \cong Q$ and if P has a w -deadlock, then Q has a w -deadlock (and vice versa, by equivalence).

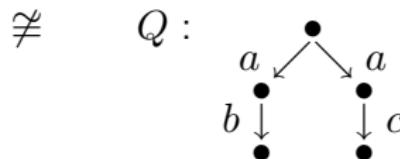
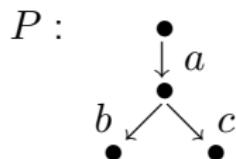
Definition 3.7 (Deadlock)

Let $P, Q \in Prc$ and $w \in Act^*$ such that $P \xrightarrow{w}^* Q$ and $Q \not\rightarrow$. Then Q is called a **w -deadlock** of P .

- Thus $P := a.b.\text{nil} + a.\text{nil}$ has an a -deadlock, in contrast to $Q := a.b.\text{nil}$.
- Such properties are important since it can be crucial that a certain communication is eventually possible.
- We therefore extend our set of postulates: our semantic equivalence \cong should
 - ① identify processes with identical LTSs;
 - ② imply trace equivalence;
 - ③ be a congruence; and
 - ④ be **deadlock sensitive**, i.e., if $P \cong Q$ and if P has a w -deadlock, then Q has a w -deadlock (and vice versa, by equivalence).

Deadlocks II

The combination of congruence and deadlock sensitivity also excludes the following equivalence:



If $P \cong Q$, by congruence this equivalence should hold in every context. But $C[\cdot] := \text{new } a, b, c (\bar{a}.\bar{b}.\text{nil} \parallel \cdot)$ yields the following conflict:



no τ -deadlock

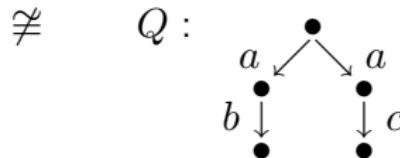
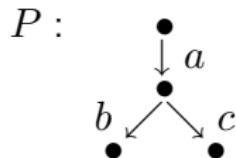


τ -deadlock

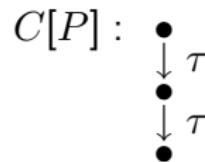
(Note: P and Q are obviously trace equivalent)

Deadlocks II

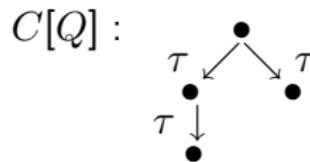
The combination of congruence and deadlock sensitivity also excludes the following equivalence:



If $P \cong Q$, by congruence this equivalence should hold in every context. But $C[\cdot] := \text{new } a, b, c (\bar{a}.\bar{b}.\text{nil} \parallel \cdot)$ yields the following conflict:



no τ -deadlock

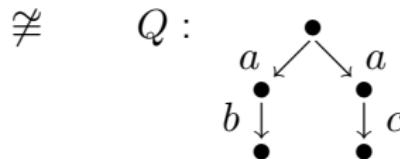
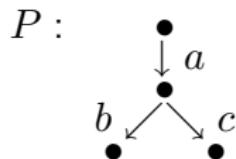


τ -deadlock

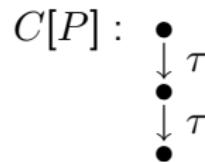
(Note: P and Q are obviously trace equivalent)

Deadlocks II

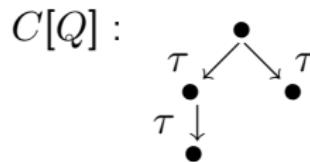
The combination of congruence and deadlock sensitivity also excludes the following equivalence:



If $P \cong Q$, by congruence this equivalence should hold in every context. But $C[\cdot] := \text{new } a, b, c (\bar{a}.\bar{b}.\text{nil} \parallel \cdot)$ yields the following conflict:



no τ -deadlock



τ -deadlock

(Note: P and Q are obviously trace equivalent)