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

Outline

Repetition: Syntax of CCS I

Definition (Syntax of CCS)

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

$$
\begin{array}{rlll}
P ::= & \mathsf{nil} & & \text{(inaction)} \\
\mid & \alpha.P & & \text{(prefixing)} \\
\mid & P_1 + P_2 & & \text{(choice)} \\
\mid & P_1 \parallel P_2 & & \text{(parallel composition)} \\
\mid & \mathsf{new}\, a\, P & & \text{(restriction)} \\
\mid & A(a_1, \ldots, a_n) & & \text{(process call)}
\end{array}
$$

where $\alpha \in Act$, $a, a_i \in N$, and $A \in Pid$.

Repetition: Syntax of CCS II

Definition (continued)

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

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

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

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

Definition (Semantics of CCS)

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

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}\ (\mathsf{Act}) \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}\ (\mathsf{Com})$$

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

$$\frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q}\ (\mathsf{Par}_1) \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'}\ (\mathsf{Par}_2)$$

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin \{a, \overline{a}\}}{\mathsf{new}\, a\, P \xrightarrow{\alpha} \mathsf{new}\, a\, P'}\ (\mathsf{New}) \qquad \frac{A(\vec{a}) = P \quad P[\vec{a} \mapsto \vec{b}] \xrightarrow{\alpha} P'}{A(\vec{b}) \xrightarrow{\alpha} P'}\ (\mathsf{Call})$$

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

Outline

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

(nested scalar recursion)

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

Disadvantage: bad readability of process definitions

Outline

Equivalence Relations

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 $s, t, u \in S$.

Equivalence of CCS Processes

- **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

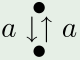
$$\text{meaning of a process } P := LTS(P)$$

- **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

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

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

CCS Congruences

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

$\Longrightarrow$ 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

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

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

Outline

Trace Equivalence I

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)

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

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

Trace Equivalence II

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:
  1. 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.
  2. it implies trace equivalence: trivial
  3. it is a congruence:

CCS Congruences [repetition]

Goal: replacing a subcomponent of a system by an equivalent process should yield an equivalent systems
$\Longrightarrow$ 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

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

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

Trace Equivalence III

Theorem 3.6

Trace equivalence is a congruence.

Proof.

(only for +; remaining operators analogously)
Clearly we have:

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

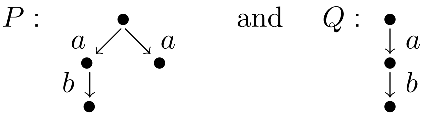
Now let $P, Q, R \in Prc$ with $Tr(P) = Tr(Q)$. Then:

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

□

Trace Equivalence IV

- 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

$\Longrightarrow$ take into account such deadlock properties

Outline

Deadlocks I

Definition 3.7 (Deadlock)

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

- Thus $P := a.b.\mathsf{nil} + a.\mathsf{nil}$ has an a-deadlock, in contrast to $Q := a.b.\mathsf{nil}$.
- 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 $C[\cdot] := \mathsf{new}\, a, b, c\, (\overline{a}.\overline{b}.\mathsf{nil} \parallel \cdot)$ yields the following conflict:

no τ-deadlock τ-deadlock

(Note: P and Q are obviously trace equivalent)