

Modeling Concurrent and Probabilistic Systems

Lecture 2: Semantics of CCS

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 of CCS

2 Semantics of CCS

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:

$P ::= \text{nil}$	(inaction)
$\quad \alpha.P$	(prefixing)
$\quad P_1 + P_2$	(choice)
$\quad P_1 \parallel P_2$	(parallel composition)
$\quad \text{new } a P$	(restriction)
$\quad A(a_1, \dots, a_n)$	(process call)

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

1 Repetition: Syntax of CCS

2 Semantics of CCS

Labeled Transition Systems

Goal: represent behavior of system by (infinite) graph

- nodes = system states
- edges = transitions between states

Definition 2.1 (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

Labeled Transition Systems

Goal: represent behavior of system by (infinite) graph

- nodes = system states
- edges = transitions between states

Definition 2.1 (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

Labeled Transition Systems

Goal: represent behavior of system by (infinite) graph

- nodes = system states
- edges = transitions between states

Definition 2.1 (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

We define the assignment

$$\begin{array}{ll} \text{syntax} & \rightarrow \text{ semantics} \\ \text{process definition} & \mapsto \text{ LTS} \end{array}$$

by induction over the syntactic structure of process expressions. Here we employ **derivation rules** of the form

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

which can be composed to complete **derivation trees**.

Definition 2.2 (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 ($P, P', Q, Q' \in Prc, \alpha \in Act, \lambda \in N \cup \bar{N}, a \in N$):

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

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

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

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

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

Example 2.3

- ❶ One-place buffer:

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

- ❷ Sequential two-place buffer:

$$B_0(in, out) = in.B_1(in, out)$$

$$B_1(in, out) = \overline{out}.B_0(in, out) + in.B_2(in, out)$$

$$B_2(in, out) = \overline{out}.B_1(in, out)$$

- ❸ Parallel two-place buffer:

$$B_{\parallel}(in, out) = \text{new } com (B(in, com) \parallel B(com, out))$$

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

(on the board)

Example 2.3 (continued)

Complete LTS of parallel two-place buffer:

$$\begin{array}{ccc}
 B_{\parallel}(in, out) & & \text{new } com(B(in, com) \parallel B(com, out)) \\
 \downarrow in \quad \swarrow in \quad \uparrow \overline{out} & & \\
 \text{new } com(\overline{com}.B(in, com) \parallel & \xrightarrow{\tau} & \text{new } com(B(in, com) \parallel \\
 B(com, out)) & & \overline{out}.B(com, out)) \\
 \nwarrow \overline{out} \quad \swarrow in & & \\
 \text{new } com(\overline{com}.B(in, com) \parallel & & \overline{out}.B(com, out))
 \end{array}$$

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: $P ::= \text{nil} \mid \dots \mid \text{fix } A P \in \text{Prec}$ (where $A \in \text{Pid}$)

- semantics:
$$\frac{P[A \mapsto P] \xrightarrow{\alpha} P'}{\text{fix } A P \xrightarrow{\alpha} \text{fix } A P'} \text{ (Fix)}$$

- example:
$$\frac{\frac{in.\overline{out}.in.\overline{out}.B \xrightarrow{in} \overline{out}.in.\overline{out}.B}{\text{fix } B in.\overline{out}.B \xrightarrow{in} \text{fix } B \overline{out}.in.\overline{out}.B} \text{ (Act)}}{\text{fix } B in.\overline{out}.B \xrightarrow{in} \text{fix } B \overline{out}.in.\overline{out}.B} \text{ (Fix)}$$

(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: $P ::= \text{nil} \mid \dots \mid \text{fix } A P \in \text{Prc}$ (where $A \in \text{Pid}$)

- semantics:
$$\frac{P[A \mapsto P] \xrightarrow{\alpha} P'}{\text{fix } A P \xrightarrow{\alpha} \text{fix } A P'} \text{ (Fix)}$$

- example:
$$\frac{\frac{in.\overline{out}.in.\overline{out}.B \xrightarrow{in} \overline{out}.in.\overline{out}.B}{\text{fix } B in.\overline{out}.B \xrightarrow{in} \text{fix } B \overline{out}.in.\overline{out}.B} \text{ (Act)}}{\text{fix } B in.\overline{out}.B \xrightarrow{in} \text{fix } B \overline{out}.in.\overline{out}.B} \text{ (Fix)}$$

(nested scalar recursion)

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

Disadvantage: bad readability of process definitions

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

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

(simultaneous recursion)

Alternative: explicit **fixpoint operator**

- syntax: $P ::= \text{nil} \mid \dots \mid \text{fix } A P \in \text{Proc}$ (where $A \in \text{Pid}$)

- semantics:
$$\frac{P[A \mapsto P] \xrightarrow{\alpha} P'}{\text{fix } A P \xrightarrow{\alpha} \text{fix } A P'} \text{ (Fix)}$$

- example:
$$\frac{\frac{in.\overline{out}.in.\overline{out}.B \xrightarrow{in} \overline{out}.in.\overline{out}.B}{\text{fix } B in.\overline{out}.B \xrightarrow{in} \text{fix } B \overline{out}.in.\overline{out}.B} \text{ (Act)}}{\text{fix } B in.\overline{out}.B \xrightarrow{in} \text{fix } B \overline{out}.in.\overline{out}.B} \text{ (Fix)}$$

(nested scalar recursion)

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

Disadvantage: bad readability of process definitions