Concurrent Object-Oriented Programming

Lecture 8: Process Calculi

Sebastian Nanz
Today’s Lecture

- A short intermezzo in the SCOOP saga
- **Goal**: To have a look at one of the foundational theoretical models of concurrency: process calculi
Introduction

Process Calculi

- **Question**: Why do we need a theoretical model of concurrent computation?
- Turing machines or the $\lambda$-calculus have proved to be useful models of sequential systems
- Abstracting away from implementation details yields general insights into programming and computation
- Process calculi help to focus on the essence of concurrent systems: interaction
The Calculus of Communicating Systems (CCS)

- We study the **Calculus of Communicating Systems (CCS)**
- Introduced by [Milner 1980]
- Milner’s general model:
  - A **concurrent system** is a collection of processes
  - A **process** is an independent agent that may perform internal activities in isolation or may interact with the environment to perform shared activities
- Milner’s insight: Concurrent processes have an algebraic structure

\[
P_1 \circ P_2 \Rightarrow P_1 \circ P_2
\]

- This is why a process calculus is sometime called a **process algebra**
Introduction

Introductory Example: A Simple Process

- A coffee and tea machine may take an order for either tea or coffee, accept the appropriate payment, pour the ordered drink, and terminate:

  \[ \text{tea}.\text{coin}.\text{cup\_of\_tea}.0 + \text{coffee}.\text{coin}.\text{coin}.\text{cup\_of\_coffee}.0 \]

- We have the following elements of syntax:
  - **Actions:** \text{tea}, \text{cup\_of\_tea}, etc.
  - **Sequential composition:** the dot “.” (first do action \text{tea}, then \text{coin}, ...)
  - **Non-deterministic choice:** the plus “+” (either do \text{tea} or \text{coffee})
  - **Terminated process:** 0
Introductory Example: Execution of a Simple Process

- When a process executes it performs some action, and becomes a new process.
- The execution of an action \( a \) is symbolized by a transition \( a \to \):

\[
\text{tea.coin.cup_of_tea.0} + \text{coffee.coin.coin.cup_of_coffee.0} \\
\xrightarrow{\text{tea}} \text{coin.cup_of_tea.0} \\
\xrightarrow{\text{coin}} \text{cup_of_tea.0} \\
\xrightarrow{\text{cup_of_tea}} 0
\]
Syntax of CCS
Syntax of CCS

Goal: In the following we introduce the syntax of CCS step-by-step

Basic principle

1. Define atomic processes that model the simplest possible behavior
2. Define composition operators that build more complex behavior from simpler ones
The Terminal Process

The simplest possible behavior is **no behavior**

**Terminal process**

We write $0$ (pronounced “nil”) for the *terminal* or *inactive process*

- $0$ models a system that is either deadlocked or has terminated
- $0$ is the only atomic process of CCS
Names and Actions

We assume an infinite set $\mathcal{A}$ of port names, and a set $\overline{\mathcal{A}} = \{ \overline{a} | a \in \mathcal{A} \}$ of complementary port names.

Input actions

When modeling we use a name $a$ to denote an input action, i.e. the receiving of input from the associated port $a$.

Output actions

We use a co-name $\overline{a}$ to denote an output action, i.e. the sending of output to the associated port $a$.

Internal actions

We use $\tau$ to denote the distinguished internal action.

The set of actions $Act$ is given by $Act = \mathcal{A} \cup \overline{\mathcal{A}} \cup \{ \tau \}$.
The simplest actual behavior is sequential behavior

**Action prefixing**

If $P$ is a process we write

$$\alpha.P$$

to denote the prefixing of $P$ with the action $\alpha$

- $\alpha.P$ models a system that is ready to perform the action, $\alpha$, and then behave as $P$, i.e.

$$\alpha.P \xrightarrow{\alpha} P$$
Example: Action Prefixing

A process that starts a timer, performs some internal computation, and then stops the timer:

$$\texttt{go.}\tau.\texttt{stop.0} \xrightarrow{\texttt{go}} \tau.\texttt{stop.0} \xrightarrow{\tau} \texttt{stop.0} \xrightarrow{\texttt{stop}} 0$$
Process Interfaces

Interfaces

The set of **input** and **output actions** that a process \( P \) may perform in isolation constitutes the interface of \( P \)

- The interface enumerates the ports that \( P \) may use to interact with the environment

**Example:** The interface of the coffee and tea machine is:

\[
tea, \text{ coffee, coin, cup_of_tea, cup_of_coffee}
\]
Syntax of CCS

**Non-deterministic Choice**

A more advanced sequential behavior is that of alternative behaviors.

**Non-deterministic choice**

If $P$ and $Q$ are processes then we write $P + Q$ to denote the non-deterministic choice between $P$ and $Q$.

$P + Q$ models a process that can either behave as $P$ (discarding $Q$) or as $Q$ (discarding $P$).
Example: Non-deterministic Choice

\[
tea.\text{coin.}\text{cup\_of\_tea}.0 + \, coffee.\text{coin.}\text{coin.}\text{cup\_of\_coffee}.0 \\
\xrightarrow{\text{tea}} \, coin.\text{cup\_of\_tea}.
\]

Note that:

- prefixing binds harder than plus and
- the choice is made by the initial coffee/tea button press
Process Constants and Recursion

The most advanced sequential behavior is the recursive behavior

Process constants
A process may be the invocation of a process constant, $K \in \mathcal{K}$
This is only meaningful if $K$ is defined beforehand

Recursive definition
If $K$ is a process constant and $P$ is a process we write

$$K \overset{\text{def}}{=} P$$

to give a recursive definition of the behavior of $K$
(recursive if $P$ invokes $K$)
Syntax of CCS

Example: Recursion (1)

A system clock, $SC$, sends out regular clock signals forever:

$$SC \overset{\text{def}}{=} tick \cdot SC$$

The system $SC$ may behave as:

$$tick \cdot SC \xrightarrow{tick} SC \xrightarrow{tick} \ldots$$
Example: Recursion (2)

A fully automatic coffee and tea machine CTM works as follows:

\[ \text{CTM} \overset{\text{def}}{=} \text{tea.coin.cup_of_tea.CTM} + \text{coffee.coin.coin.cup_of_coffee.CTM} \]

The system CTM may e.g. do:

\[ \text{tea.coin.cup_of_tea.CTM} + \text{coffee.coin.coin.cup_of_coffee.CTM} \]

\[
\begin{align*}
\text{tea} & \rightarrow \text{coin.cup_of_tea.CTM} \\
\text{coin} & \rightarrow \text{cup_of_tea.CTM} \\
\text{cup_of_tea} & \rightarrow \text{CTM} \\
\alpha & \rightarrow \ldots
\end{align*}
\]

This will serve drinks ad infinitum
Finally: concurrent behavior

Parallel composition

If $P$ and $Q$ are processes we write

$$ P \parallel Q $$

to denote the parallel composition of $P$ and $Q$

- $P \parallel Q$ models a process that behaves like $P$ and $Q$ in parallel:
  - Each may proceed independently
  - If $P$ is ready to perform an action $a$ and $Q$ is ready to perform the complementary action $\bar{a}$, they may interact
Syntax of CCS

Example: Parallel Composition

Recall the coffee and tea machine:

\[
\text{CTM} \overset{\text{def}}{=} \text{tea.\text{coin.\text{cup}\_of\_tea.\text{CTM}}} + \text{coffee.\text{coin.\text{coin.\text{cup}\_of\_coffee.\text{CTM}}}}
\]

Now consider the regular customer – the Computer Scientist, CS:

\[
\text{CS} \overset{\text{def}}{=} \text{tea.\text{coin.\text{cup}\_of\_tea.\text{teach.\text{CS}}}} + \text{coffee.\text{coin.\text{coin.\text{cup}\_of\_coffee.\text{publish.\text{CS}}}}}
\]

- CS must drink coffee to publish
- CS can only teach on tea
Syntax of CCS

Example: Parallel Composition

On an average Tuesday morning the system

$$\text{CTM} | \text{CS}$$

is likely to behave as follows:

$$(\text{tea}.\text{coin}.\text{cup}_\text{of}_\text{tea}.\text{CTM} + \text{coffee}.\text{coin}.\text{coin}.\text{cup}_\text{of}_\text{coffee}.\text{CTM})$$

$$| (\text{tea}.\text{coin}.\text{cup}_\text{of}_\text{tea}.\text{teach}.\text{CS} + \text{coffee}.\text{coin}.\text{coin}.\text{cup}_\text{of}_\text{coffee}.\text{publish}.\text{CS})$$

$$\xrightarrow{\tau} (\text{coin}.\text{cup}_\text{of}_\text{tea}.\text{CTM}) | (\text{coin}.\text{cup}_\text{of}_\text{tea}.\text{teach}.\text{CS})$$

$$\xrightarrow{\tau} (\text{cup}_\text{of}_\text{tea}.\text{CTM}) | (\text{cup}_\text{of}_\text{tea}.\text{teach}.\text{CS})$$

$$\xrightarrow{\tau} \text{CTM} | (\text{teach}.\text{CS})$$

$$\text{teach} \xrightarrow{\tau} \text{CTM} | \text{CS}$$

- Note that the synchronisation of actions such as \text{tea}/\text{tea} is expressed by a $\tau$-action (i.e. regarded as an internal step)
Restriction

We control unwanted interactions with the environment by restricting the scope of port names

Restriction

if $P$ is a process and $A$ is a set of port names we write

$$P \setminus A$$

for the restriction of the scope of each name in $A$ to $P$

- Removes each name $a \in A$ and the corresponding co-name $\overline{a}$ from the interface of $P$
- Makes each name $a \in A$ and the corresponding co-name $\overline{a}$ inaccessible to the environment
Example: Restriction

- Recall the coffee and tea machine and the computer scientist:
  \[ \text{CTM} \mid \text{CS} \]

- Restricting the coffee and tea machine on *coffee* makes the *coffee*-button inaccessible to the computer scientist:
  \[ (\text{CTM} \setminus \{\text{coffee}\}) \mid \text{CS} \]

- As a consequence *CS* can only teach, and never publish
Syntax of CCS

Summary: Syntax of CCS

\[ P ::= \begin{align*} & K \quad \mid \quad \text{process constants} \ (K \in \mathcal{K}) \\ & \alpha.P \quad \mid \quad \text{prefixing} \ (\alpha \in \mathcal{Act}) \\ & \sum_{i \in I} P_i \quad \mid \quad \text{summation} \ (I \text{ is an arbitrary index set}) \\ & P_1 | P_2 \quad \mid \quad \text{parallel composition} \\ & P \setminus L \quad \mid \quad \text{restriction} \ (L \subseteq \mathcal{A}) \end{align*} \]

The set of all terms generated by the abstract syntax is called CCS process expressions.

Notation

\[ P_1 + P_2 = \sum_{i \in \{1,2\}} P_i \quad \quad \text{Nil} = 0 = \sum_{i \in \emptyset} P_i \]
CCS Program

A collection of defining equations of the form

\[ K \overset{\text{def}}{=} P \]

where \( K \in \mathcal{K} \) is a process constant and \( P \in \mathcal{P} \) is a CCS process expression

- Only one defining equation per process constant
- Recursion is allowed: e.g. \( A \overset{\text{def}}{=} \overline{a}.A \mid A \)
- Note that the program itself gives only the definitions of process constants: we can only execute processes (which can however mention the process constants defined in the program)
Exercise: Syntax of CCS

Which of the following expressions are correctly built CCS expressions? Assume that $A$, $B$ are process constants and that $a$, $b$ are port names.

- $a.b.A + B$ ✓
- $(a.0 + \overline{a}.A) \setminus \{a, b\}$ ✓
- $(a.0 | \overline{a}.A) \setminus \{a, \tau\}$ x
- $\tau.\tau.B + 0$ ✓
- $(a.b.A + \overline{a}.0)|B$ ✓
- $(a.b.A + \overline{a}.0).B$ x
Operational Semantics of CCS
Operational Semantics of CCS

Operational Semantics

▶ **Goal**: Formalize the execution of a CCS process

**Syntax**

CCS

(process term + equations)

**Semantics**

LTS

(labelled transition systems)
Labelled Transition System

Definition
A labelled transition system (LTS) is a triple \((\text{Proc}, \text{Act}, \{\alpha \rightarrow | \alpha \in \text{Act}\})\) where

- \(\text{Proc}\) is a set of processes (the states),
- \(\text{Act}\) is a set of actions (the labels), and
- for every \(\alpha \in \text{Act}\), \(\alpha \rightarrow \subseteq \text{Proc} \times \text{Proc}\) is a binary relation on processes called the transition relation.

We use the infix notation \(P \xrightarrow{\alpha} P'\) to say that \((P, P') \in \alpha \rightarrow\). It is customary to distinguish the initial process (the start state).
Labelled Transition Systems

Conceptually it is often beneficial to think of a (finite) LTS as something that can be drawn as a directed (process) graph

- **Processes** are the nodes
- **Transitions** are the edges

**Example:** The LTS

\[
\{\{P, Q, R\}, \{a, b, \tau\}, \{P \xrightarrow{a} Q, P \xrightarrow{b} R, Q \xrightarrow{\tau} R\}\}
\]

corresponds to the graph

- **Question:** How can we produce an LTS (semantics) of a process term (syntax)?
Informal Translation

- **Terminal process**: $0$
  
  behavior: $0 \not\rightarrow$

- **Action prefixing**: $\alpha.P$
  
  behavior: $\alpha.P \xrightarrow{\alpha} P$

- **Non-deterministic choice**: $\alpha.P + \beta.Q$
  
  behavior: $P \xleftarrow{\alpha} \alpha.P + \beta.Q \xrightarrow{\beta} Q$

- **Recursion**: $X \overset{\text{def}}{=} \ldots \alpha.X$
  
  behavior: $X \xleftrightarrow{\alpha} \alpha.X$
Informal Translation

- Parallel composition: $\alpha.P | \beta.Q$
  Combines sequential composition and choice to obtain interleaving

- What about interaction?
Operational Semantics of CCS

Process Interaction

- Concurrent processes, i.e. \( P \) and \( Q \) in \( P \parallel Q \), may interact where their interfaces are compatible.
- A synchronizing interaction between two processes (sub-systems), \( P \) and \( Q \), is an activity that is internal to \( P \parallel Q \).
- Parallel composition: \( \alpha.P \parallel \beta.Q \)
  Allows interaction if \( \beta = \overline{\alpha} \)

\[
P \parallel \overline{a}.Q
\]

\[
\begin{array}{c}
a \\
\overline{a}
\end{array}
\]

behavior: \( a.P \parallel \overline{a}.Q \xrightarrow{\tau} P \parallel Q \)

\[
\begin{array}{c}
\overline{a} \\
\overline{a} \\
a
\end{array}
\]

\( a.P \parallel Q \)
Structural Operational Semantics for CCS

Structural Operational Semantics (SOS) [Plotkin 1981]
Small-step operational semantics where the behavior of a system is inferred using syntax driven rules

Given a collection of CCS defining equations, we define the following LTS \((Proc, Act, \{\xrightarrow{a} \mid a \in Act\})\):

- \(Proc\) is the set of all CCS process expressions
- \(Act\) is the set of all CCS actions including \(\tau\)
- the transition relation is given by SOS rules of the form:

\[
\text{RULE} \quad \text{premises} \quad \text{conclusion} \quad \text{conditions}
\]
SOS rules for CCS

\[
\begin{align*}
\text{ACT} & : \quad \alpha . P \xrightarrow{\alpha} P \\
\text{SUM}_j & : \quad \sum_{i \in I} P_i \xrightarrow{\alpha} P'_j \quad j \in I \\
\text{COM1} & : \quad P \xrightarrow{\alpha} P' \quad P \parallel Q \xrightarrow{\alpha} P'|Q \\
\text{COM2} & : \quad Q \xrightarrow{\alpha} Q' \quad P \parallel Q \xrightarrow{\alpha} P'|Q' \\
\text{COM3} & : \quad P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q' \quad P \parallel Q \xrightarrow{\tau} P'|Q' \\
\text{RES} & : \quad P \xrightarrow{\alpha} P' \quad P \setminus L \xrightarrow{\alpha} P' \setminus L \\
\text{CON} & : \quad P \xrightarrow{\alpha} P' \quad K \xrightarrow{\alpha} P' \quad K \overset{\text{def}}{=} P
\end{align*}
\]
Exercise: Derivations

Let $A \overset{\text{def}}{=} a.A$. Show that

$$((A|\overline{a}.0)|b.0) \xrightarrow{a} ((A|\overline{a}.0)|b.0).$$
Restriction and Interaction

Restriction can be used to produce closed systems, i.e. their actions can only be taken internally (visible as $\tau$-actions)
Behavioral Equivalence
Goal: Express the notion that two concurrent systems “behave in the same way”

We are not interested in syntactical equivalence, but only in the fact that the processes have the same behavior

Main idea: two processes are behaviorally equivalent if and only if an external observer cannot tell them apart

Bisimulation [Park 1980]: Two processes are equivalent if they have the same traces and the states that they reach are also equivalent
Strong Bisimilarity

Let \((\text{Proc}, \text{Act}, \{ \frac{\alpha}{\rightarrow} | \alpha \in \text{Act} \})\) be an LTS

Strong Bisimulation

A binary relation \(R \subseteq \text{Proc} \times \text{Proc}\) is a strong bisimulation iff whenever \((P, Q) \in R\) then for each \(\alpha \in \text{Act}:\)

- if \(P \xrightarrow{\alpha} P'\) then \(Q \xrightarrow{\alpha} Q'\) for some \(Q'\) such that \((P', Q') \in R\)
- if \(Q \xrightarrow{\alpha} Q'\) then \(P \xrightarrow{\alpha} P'\) for some \(P'\) such that \((P', Q') \in R\)

Strong Bisimilarity

Two processes \(P_1, P_2 \in \text{Proc}\) are strongly bisimilar \((P_1 \sim P_2)\) if and only if there exists a strong bisimulation \(R\) such that \((P_1, P_2) \in R\)

\[
\sim = \bigcup \{ R \mid R \text{ is a strong bisimulation} \} 
\]
Strong Bisimilarity of CCS Processes

- The concept of strong bisimilarity is defined for LTS
- The semantics of CCS is given in terms of LTS, whose states are CCS processes
- Thus, the definition also applies to CCS processes
  - Two processes are bisimilar if there is a concrete strong bisimulation relation that relates them
  - To show that two processes are bisimilar it suffices to exhibit such a concrete relation
Consider the processes $P$ and $Q$ with the following behavior:

We claim that they are bisimilar.
Example: Strong Bisimulation

To show our claim we exhibit the following strong bisimulation relation:

\[ R = \{(P, Q), (P_1, Q_1), (P_2, Q_1)\} \]

- (P, Q) is in \( R \)
- \( R \) is a bisimulation:
  - For each pair of states in \( R \), all possible transitions from the first can be matched by corresponding transitions from the second
  - For each pair of states in \( R \), all possible transitions from the second can be matched by corresponding transitions from the first
Example: Strong Bisimulation

Graphically, we show $\mathcal{R}$ with dotted lines:

Now it is easy to see that:

- For each pair of states in $\mathcal{R}$, all possible transitions from the first can be matched by corresponding transitions from the second.
- For each pair of states in $\mathcal{R}$, all possible transitions from the second can be matched by corresponding transitions from the first.
Consider the processes

\[ P \overset{\text{def}}{=} a.(b.0 + c.0) \]
\[ Q \overset{\text{def}}{=} a.b.0 + a.c.0 \]

and show that \( P \not\sim Q \)
Discussion
We have seen CCS, a so-called process calculus.

Process calculi are models of concurrency that help focus on some fundamental questions, e.g. “when can two concurrent systems be considered the same?” (behavioral equivalence).

In order to be able to formulate such questions precisely, all aspects of CCS were formally defined.

In particular, there were rules to map a concurrent system described in CCS (a piece of syntax) into a labelled transition system describing its behavior (the semantics of the system).
While CCS is one of the fundamental models of concurrency, it is not the only one.

The reason for this is that there are many forms of concurrency one might like to describe.

For example, the $\pi$-calculus [Milner, Parrow, Walker 1992] also takes mobility into account.

CCS is a model of concurrency, not a programming language – for “everyday use” too many details are abstracted away.

However, the formal techniques studied in CCS can help to design better programming languages as well.