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

- We study the Calculus of Communicating Systems (CCS) [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

  \[
  \begin{array}{c}
  P_1 \text{ op } P_2 \\
  \Rightarrow \\
  P_1 \text{ op } P_2
  \end{array}
  \]

- This is why a process calculus is sometime called a process algebra
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:

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

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

\[
\begin{align*}
tea &\cdot coin \cdot cup\_of\_tea.0 + coffee &\cdot coin &\cdot coin &\cdot cup\_of\_coffee.0 \\
\quad \xrightarrow{tea} &\quad coin &\cdot cup\_of\_tea.0 \\
\quad \xrightarrow{coin} &\quad cup\_of\_tea.0 \\
\quad \xrightarrow{cup\_of\_tea} &\quad 0
\end{align*}
\]
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.
- 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 $\bar{\mathcal{A}} = \{\bar{a} \mid a \in \mathcal{A}\}$ 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 $\bar{a}$ 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 $\text{Act}$ is given by $\text{Act} = \mathcal{A} \cup \bar{\mathcal{A}} \cup \{\tau\}$.
Action prefixing

- 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 behaves 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:

$$\overline{go}.\tau.\overline{stop}.0 \xrightarrow{go} \tau.\overline{stop}.0 \xrightarrow{\tau} \overline{stop}.0 \xrightarrow{stop} 0$$
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, coffee, coin, cup_of_tea, cup_of_coffee
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

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

- **Note**
  - Prefixing binds harder than plus
  - The choice is made by the initial *coffee / tea* button press
Process constants and recursion

- The most advanced sequential behavior is 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 $K$).
Example: Recursion (1)

- A system clock, \( SC \), sends out regular clock signals forever:

\[
SC \overset{\text{def}}{=} \text{tick}.SC
\]

- The system \( SC \) may behave as:

\[
\text{tick}.SC \xrightarrow{\text{tick}} SC \xrightarrow{\text{tick}} \ldots
\]
Example: Recursion (2)

- A fully automatic coffee and tea machine CTM

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

- The system CTM may e.g. do:

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

\[ \text{tea} \rightarrow \text{coin.cup_of.tea.CTM} \]

\[ \text{coin} \rightarrow \text{cup_of.tea.CTM} \]

\[ \text{cup_of.tea} \rightarrow \text{CTM} \]

\[ \alpha \rightarrow \ldots \]

- This will serve drinks ad infinitum
Parallel composition

- Finally: concurrent behavior
- Parallel composition
  - If $P$ and $Q$ are processes we write
    \[
    P \mid Q
    \]
    to denote the parallel composition of $P$ and $Q
  - $P \mid Q$ 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 $\overline{a}$, they may interact
Example: Parallel composition

- Recall the coffee and tea machine:
  \[
  \text{CTM} \overset{\text{def}}{=} \text{tea.coin.cup\_of\_tea.CTM} + \text{coffee.coin.coin.cup\_of\_coffee.CTM}
  \]

- Now consider a regular customer, the Computer Scientist CS:
  \[
  \text{CS} \overset{\text{def}}{=} \text{tea.coin.cup\_of\_tea.teach.CS} + \text{coffee.coin.coin.cup\_of\_coffee.publish.CS}
  \]

- CS must drink coffee to publish
- CS can only teach on tea
Example: Parallel composition

- On an average Tuesday morning the system

\[ \text{CTM} \mid \text{CS} \]

is likely to behave as follows:

\[
(\text{tea.coin.cup}\_\text{of}\_\text{tea} \cdot \text{CTM} + \text{coffee.coin.coin.cup}\_\text{of}\_\text{coffee} \cdot \text{CTM}) \\
\mid \hspace{1em} (\text{tea.cup}\_\text{of}\_\text{tea}.\text{teach} \cdot \text{CS} + \text{coffee.coin.cup}\_\text{of}\_\text{coffee}.\text{publish} \cdot \text{CS})
\]

\[\tau \rightarrow (\text{coin.cup}\_\text{of}\_\text{tea} \cdot \text{CTM}) \mid (\text{coin.cup}\_\text{of}\_\text{tea}.\text{teach} \cdot \text{CS})\]

\[\tau \rightarrow (\text{cup}\_\text{of}\_\text{tea} \cdot \text{CTM}) \mid (\text{cup}\_\text{of}\_\text{tea}.\text{teach} \cdot \text{CS})\]

\[\tau \rightarrow \text{CTM} \mid (\text{teach} \cdot \text{CS})\]

\[\text{teach} \rightarrow \text{CTM} \mid \text{CS}\]

- Note that the synchronisation of actions such as \text{tea / 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
Summary: Syntax of CCS

\[ P ::= K \quad | \quad \alpha.P \quad | \quad \sum_{i \in I} P_i \quad | \quad P_1|P_2 \quad | \quad P \setminus L \]

- process constants \((K \in \mathcal{K})\)
- prefixing \((\alpha \in \text{Act})\)
- summation \((I \) is an arbitrary index set\)
- parallel composition
- restriction \((L \subseteq \mathcal{A})\)

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

- Notation

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

- 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. \( A \overset{\text{def}}{=} \overline{a}.A \mid A \)

- 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

\[
\begin{align*}
  a.b.A + B & \quad \checkmark \\
  (a.0 + \overline{a}.A) \setminus \{a, b\} & \quad \checkmark \\
  (a.0 | \overline{a}.A) \setminus \{a, \tau\} & \quad \times \\
  \tau.\tau.B + 0 & \quad \checkmark \\
  (a.b.A + \overline{a}.0) | B & \quad \checkmark \\
  (a.b.A + \overline{a}.0).B & \quad \times
\end{align*}
\]
Operational Semantics of CCS
Operational semantics

- Goal: Formalize the execution of a CCS process

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCS</td>
<td>LTS</td>
</tr>
<tr>
<td>(process term + equations)</td>
<td>(labelled transition systems)</td>
</tr>
</tbody>
</table>
Labelled transition systems

- A labelled transition system (LTS) is a triple $(\text{Proc}, \text{Act}, \{(\alpha \rightarrow | \alpha \in \text{Act})\})$ where
  - Proc is a set of processes (the states),
  - 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

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

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: $X \overset{\text{def}}{=} \cdots \alpha.X$
  
  behavior: 
  
  \begin{align*}
  &X \xrightarrow{\alpha} \alpha.X \\
  \cdots
  
  &X \xleftarrow{\alpha} \alpha.X
  
  \end{align*}
Informal translation

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

What about interaction?
Process interaction

- Concurrent processes, i.e. $P$ and $Q$ in $P | Q$, 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 $P | Q$.
- Parallel composition: $\alpha.P | \beta.Q$
- Allows interaction if $\beta = \overline{\alpha}$

$P | \overline{a}.Q$

$\alpha.P | \overline{a}.Q \xrightarrow{\tau} P | Q$

behavior:

$P | \overline{a}.Q$

$\alpha.P | Q$

$a$  $\overline{a}$

$\overline{a}$  $a$
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 LTS \((\mathit{Proc}, \mathit{Act}, \{ \xrightarrow{a} \mid a \in \mathit{Act} \})\)
  - \(\mathit{Proc}\) is the set of all CCS process expressions
  - \(\mathit{Act}\) is the set of all CCS actions including
  - the transition relation is given by SOS rules of the form:

  ```
  RULE \begin{array}{c}
  \text{premises} \\
  \hline
  \text{conclusion} \\
  \text{conditions}
  \end{array}
  ```
SOS rules for CCS

**ACT**

\[
\frac{\alpha \cdot P \xrightarrow{\alpha} P}{P}
\]

**SUM**

\[
\frac{P_j \xrightarrow{\alpha} P_j'}{\sum_{i \in I} P_i \xrightarrow{\alpha} P_j'} \quad j \in I
\]

**COM1**

\[
\frac{P \xrightarrow{\alpha} P'}{P | Q \xrightarrow{\alpha} P' | Q}
\]

**COM2**

\[
\frac{Q \xrightarrow{\alpha} Q'}{P | Q \xrightarrow{\alpha} P | Q'}
\]

**COM3**

\[
\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P | Q \xrightarrow{\tau} P' | Q'}
\]

**RES**

\[
\frac{P \xrightarrow{\alpha} P'}{P \ \ \alpha \not\in L}
\]

**CON**

\[
\frac{K \xrightarrow{\alpha} P'}{K \overset{\text{def}}{=} P}
\]
Example: Derivations

- Let $A \overset{\text{def}}{=} a.A$. Show that $((A \mid \overline{a}.0) \mid b.0) \overset{a}{\longrightarrow} ((A \mid \overline{a}.0) \mid b.0)$
Restriction and interaction

- Restriction can be used to produce closed systems, i.e. their actions can only be taken internally (visible as \( \tau \)-actions)
Conclusion

- Process calculi provide models of concurrency, not programming languages – for “everyday use” too many details are abstracted away.

- However, the formal techniques studied in process calculi can help to design better concurrent programming languages.