Introduction

Motivations

Principles in CCS design

Syntax and Operational Semantics of CCS

Syntax

Labeled transition System

What equivalence for CCS?

Why a Calculus for Concurrency?

- The Calculus for Communicating Systems (CCS) was developed by R. Milner around the 80's.
- Other Process Calculi were proposed at about the same time: the Theory of Communicating Sequential Processes by T. Hoare and the Algebra of Communicating Processes by J. Bergstra and J. W. Klop.
- Researchers were looking for a calculus with few, orthogonal mechanisms, able to represent all the relevant concepts of concurrent computations. More complex mechanisms should be built by using the basic ones.
  - To help understanding / reasoning about / developing formal tools for concurrency.
  - To play a role, for concurrency, like that of the $\lambda$-calculus for sequential computation.

Inadequacy of standard models of computations

The $\lambda$ calculus, the Turing machines, etc. are computationally complete, yet do not capture the features of concurrent computations like
  - Interaction and communication
  - Inadequacy of functional denotation
  - Nondeterminism

Note: nondeterminism in concurrency is different from the nondeterminism used in Formal Languages, like for instance the Nondeterministic Turing Machines.
A few words about nondeterminism

In standard computation theory, if we want to compute the partial function $f$ s.t. $f(0) = 1$, a Turing Machine like this one is considered OK.

However, we would not be happy with a coffee machine that behaves in the same way.

Nondeterminism in sequential models

- Convenient tool for solving certain problems in an easy way or for characterizing complexity classes (examples: search for a path in a graph, search for a proof etc.)
- Examples of nondeterministic formalisms:
  - The nondeterministic Turing machines
  - Logic languages like Prolog and $\land$ Prolog
- The characteristics of nondeterminism in this setting:
  - It can be eliminated without loss of computational power by using backtracking.
  - Failures don’t matter: all what we are interested on is the existence of successful computations. A failure is reported only if all possible alternatives fail.

Nondeterminism in concurrent models

- Nondeterminism may arise because of interaction between processes.
- The characteristics of nondeterminism in this setting:
  - It cannot be avoided. At least, not without loosing essential parts of expressive power. All interesting models of concurrency cope with nondeterminism.
  - Failures do matter. Chosing the wrong branch might bring to an "undesirable situation". Backtracking is usually not applicable (or very costly), because the control is distributed: we should restart not one but several processes.
- Hence controlling nondeterminism is very important. In sequential programming is just a matter of efficiency, here is a matter of avoiding getting stuck in a wrong situation.
The basic kind of interaction (1/2)

- A calculus should contain only the primary constructs. For instance, the primary form of interaction. **But what is the primary form of interaction?**
- In general, concurrent languages can offer various kinds of communication. For instance:
  - Communications via shared memory.
  - Communication via channels.
  - Communication via broadcasting.
- and we could make even more distinctions
  - one-to-one / one-to-many
  - Ordered / unordered (i.e. queues / bags)
  - Bounded / unbounded.

So what is the basic kind of communication? For CCS the answer was: **none of the above!**

Example: $P$ and $Q$ communicating via a buffer $B$

In CCS, the fundamental model of interaction is **synchronous** and **symmetric**, i.e. the partners act at the same time performing complementary actions.

This kind of interaction is called **handshaking**: the partners agree simultaneously on performing the two (complementary) actions.

In Java there is a separation between active objects (threads) and passive objects (resources). CCS avoids this separation: Every (non-elementary) entity is a process.

For instance, consider two processes $P$ and $Q$ communicating via a buffer $B$. In CCS also $B$ is a process and the communication is between $P$ and $B$, and between $Q$ and $B$.

Syntax of CCS

- (channel, port) names: $a, b, c, \ldots$
- co-names: $\bar{a}, \bar{b}, \bar{c}, \ldots$ Note: $\bar{a} = a$
- silent action: $\tau$
- actions, prefixes: $\mu ::= a | \bar{a} | \tau$
- processes: $P, Q ::= 0 | \mu.P | P \parallel Q | P + Q | (\nu a)P | \text{rec}_K P$
  - $P \parallel Q$: parallel
  - $P + Q$: (external) choice
  - $(\nu a)P$: restriction
  - $\text{rec}_K P$: process $P$ with definition $K = P$
  - $\mu.P$: prefix
  - $0$: inaction
  - $\text{rec}_K P$: (defined) process name
Labeled transition system

- The semantics of CCS is defined by in terms of a labeled transition system, which is a set of triples of the form
  \[ P \xrightarrow{\mu} Q \]

  Meaning: \( P \) evolves into \( Q \) by making the action \( \mu \).

- The presence of the label \( \mu \) allows us to keep track of the interaction capabilities with the environment.

**Some examples**

- The restriction can be used to enforce synchronization.
- The parallel operator may cause infinitely many different states.
- The fragment of the calculus without parallel operator generates only finite automata / regular trees.

**Structural operational semantics**

The transitions of CCS are defined by a set of inductive rules. The system is also called structural semantics because the evolution of a process is defined in terms of the evolution of its components.

\[
\begin{align*}
\text{[Act]} & \quad \frac{P \xrightarrow{\mu} Q}{\mu.P \xrightarrow{\mu} \mu.P} \\
\text{[Res]} & \quad \frac{P \xrightarrow{\mu} Q}{P.R \xrightarrow{\mu} Q} \\
\text{[Sum1]} & \quad \frac{P \xrightarrow{\mu} P'}{P+Q \xrightarrow{\mu} P'+Q} \\
\text{[Sum2]} & \quad \frac{Q \xrightarrow{\nu} Q'}{P+Q \xrightarrow{\nu} P'+Q'} \\
\text{[Par1]} & \quad \frac{P \xrightarrow{\mu} P'}{P.Q \xrightarrow{\mu} P'.Q} \\
\text{[Par2]} & \quad \frac{Q \xrightarrow{\nu} Q'}{P.Q \xrightarrow{\nu} P.Q'} \\
\text{[Com]} & \quad \frac{P \xrightarrow{\mu} P'}{P.\sigma P' \xrightarrow{\mu} P'.\sigma P} \\
\end{align*}
\]

**Motivation**

- It is important to define formally when two system can be considered equivalent.
- There may be various "interesting" notion of equivalence, it depends on what we want (which observables we want to preserve).
- A good notion of equivalence should be a congruence, so to allow modular verification.
Examples: possible definitions of a coffee machine

- $\text{rec}_K \text{coin.(coffee.cup.K + tea.tcup.K)}$
- $\text{coin.rec}_K (\text{coffee.cup.coin.K + tea.tcup.coin.K})$
- $\text{rec}_K (\text{coin.coffee.cup.K + coin.tea.tcup.K})$

Question: which of these machines can we safely consider equivalent?

Note that these machines have all the same traces.

Exercises

- Define in CCS a semaphore with initial value $n$
- Show that maximal trace equivalence is not a congruence in CCS. By maximal traces here we mean the traces of all possible (finite or infinite) maximal runs.