Modelling and Analysing Concurrent Systems

RIO 2023 Summer School of Informatics
Rio Cuarto, Argentina; February 13–17, 2023

Lecture 1: Milner’s Calculus of Communicating Systems

Thomas Noll
Software Modelling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ws-22-23/rio/
Outline of Lecture 1

Preliminaries

Concurrency and Interaction
A Closer Look at Memory Models
A Closer Look at Reactive Systems
Overview of the Course
The Approach
Syntax of CCS
Intuitive Meaning and Examples
Formal Semantics of CCS
Infinite State Spaces
The CAAL Tool
About me

- Associate professor at the **Software Modelling and Verification Group (MOVES)** in the Department of Computer Science at RWTH Aachen University

- Research interests:
  - Reliability, Safety and Security of Hardware/Software Systems
  - Static Program Analysis for Software Optimisation and Verification
  - Formal Verification of Artificial Neural Networks

- Teaching activities:
  - Courses on *Concurrency Theory*
  - Courses on *Semantics and Verification of Software*
  - Courses on *Compiler Construction*
  - Courses on *Static Program Analysis*
  - Bridging courses on *Foundations of Informatics*
  - Seminars on advanced topics
  - Supervision of Bachelor’s and Master’s theses
About me

- Associate professor at the Software Modelling and Verification Group (MOVES) in the Department of Computer Science at RWTH Aachen University

- Research interests:
  - Reliability, Safety and Security of Hardware/Software Systems
  - Static Program Analysis for Software Optimisation and Verification
  - Formal Verification of Artificial Neural Networks

- Teaching activities:
  - Courses on Concurrency Theory
  - Courses on Semantics and Verification of Software
  - Courses on Compiler Construction
  - Courses on Static Program Analysis
  - Bridging courses on Foundations of Informatics
  - Seminars on advanced topics
  - Supervision of Bachelor’s and Master’s theses
Course Objectives

Objectives

- Understand the foundations of concurrent systems
- Understand the main semantical underpinnings of concurrency
- Model, reason about, and compare concurrent systems in a rigorous manner
## Course Objectives

### Objectives

- Understand the **foundations of concurrent systems**
- Understand the main **semantical underpinnings** of concurrency
- **Model, reason about, and compare** concurrent systems in a **rigorous** manner

### Motivation

- Supporting the **design phase** of systems
  - “Programming Concurrent Systems”
  - synchronisation, scheduling, semaphores, ...

---

4 of 37  Modelling and Analysing Concurrent Systems  
RIO 2023  
Lecture 1: Milner’s Calculus of Communicating Systems
Course Objectives

Objectives

- Understand the foundations of concurrent systems
- Understand the main semantical underpinnings of concurrency
- Model, reason about, and compare concurrent systems in a rigorous manner

Motivation

- Supporting the design phase of systems
  - “Programming Concurrent Systems”
  - synchronisation, scheduling, semaphores, ...
- Verifying functional correctness properties
  - “Model Checking”
  - validation of mutual exclusion, fairness, absence of deadlocks, ...
## Course Objectives

### Objectives
- Understand the **foundations of concurrent systems**
- Understand the main **semantical underpinnings** of concurrency
- Model, reason about, and compare concurrent systems in a rigorous manner

### Motivation
- Supporting the **design phase** of systems
  - “Programming Concurrent Systems”
  - synchronisation, scheduling, semaphores, ...
- Verifying **functional correctness properties**
  - “Model Checking”
  - validation of mutual exclusion, fairness, absence of deadlocks, ...
- Comparing expressivity of **models of concurrency**
  - “interleaving” vs. “true concurrency”
  - equivalence, refinement, abstraction, ...
Organisation of the Course

Organisation

- All material (slides, exercises, ...) made available via https://moves.rwth-aachen.de/teaching/ws-22-23/rio/
- Schedule: Mon Feb 13 – Thu Feb 16, 10:30 – 13:00
- Exam Fri Feb 17 morning
Outline of Lecture 1

Preliminaries

Concurrency and Interaction

A Closer Look at Memory Models

A Closer Look at Reactive Systems

Overview of the Course

The Approach

Syntax of CCS

Intuitive Meaning and Examples

Formal Semantics of CCS

Infinite State Spaces

The CAAL Tool
Concurrent Systems by Example

**Observation:** concurrency introduces new phenomena

**Example 1.1**

\[
x := 0; \\
(x := x + 1 \parallel x := x + 2)
\]

- At first glance: \(x\) is assigned 3
- But: both parallel components could read \(x\) before it is written
- Thus: \(x\) is assigned 2,
- If exclusive access to shared memory and atomic execution of assignments guaranteed
  \[\Rightarrow\] only possible outcome: 3
Concurrent and Interaction by Example

Observation: concurrency introduces new phenomena

Example 1.1

\[ x := 0; \]
\[ (x := x + 1 \parallel x := x + 2) \]

- At first glance: \( x \) is assigned 3
Observation: concurrency introduces new phenomena

Example 1.1

\[
x := 0; \\
(x := x + 1 \parallel x := x + 2)
\]

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
Concurrent System by Example

**Observation**: concurrency introduces new phenomena

**Example 1.1**

\[
x := 0; \\
(x := x + 1 \parallel x := x + 2)\]

value of \(x\): 0

- At first glance: \(x\) is assigned 3
- But: both parallel components could read \(x\) before it is written
Observation: concurrency introduces new phenomena

Example 1.1

\[ x := 0; \]
\[ (x := x + 1 \parallel x := x + 2) \]

value of \( x \): 0

1

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
**Concurrency and Interaction by Example**

**Observation:** concurrency introduces new phenomena

**Example 1.1**

\[
\begin{align*}
x &:= 0; \\
(x := x + 1 || x := x + 2) &\quad \text{value of } x: 0 \\
1 &\quad 2
\end{align*}
\]

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
Observation: concurrency introduces new phenomena

Example 1.1

\[ x := 0; \]
\[ (x := x + 1 \parallel x := x + 2) \]

value of \( x \): 1

1 2

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
Observation: concurrency introduces new phenomena

Example 1.1

\[ x := 0; \]
\[ (x := x + 1 \parallel x := x + 2) \quad \text{value of } x: 2 \]

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
- Thus: \( x \) is assigned 2,
Concurrency and Interaction by Example

Observation: concurrency introduces new phenomena

Example 1.1

\[ x := 0; \]
\[ (x := x + 1 \parallel x := x + 2) \]

value of \( x \): 0

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
- Thus: \( x \) is assigned 2,
Observation: concurrency introduces new phenomena

Example 1.1

\[
x := 0; \\
(x := x + 1 || x := x + 2) \quad \text{value of } x: 0 \\
1
\]

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
- Thus: \( x \) is assigned 2,
Observation: concurrency introduces new phenomena

Example 1.1

\[ x := 0; \]
\[ (x := x + 1 \parallel x := x + 2) \]

value of \(x\): 0

\[ 1 \quad 2 \]

- At first glance: \(x\) is assigned 3
- But: both parallel components could read \(x\) before it is written
- Thus: \(x\) is assigned 2,
Concurrent and Interaction by Example

**Observation:** concurrency introduces new phenomena

**Example 1.1**

\[
x := 0; \\
(x := x + 1 \parallel x := x + 2) \quad \text{value of } x: 2
\]

- At first glance: \(x\) is assigned 3
- But: both parallel components could read \(x\) before it is written
- Thus: \(x\) is assigned 2,
Observation: concurrency introduces new phenomena

Example 1.1

\[
x := 0; \\
(x := x + 1 \ || \ x := x + 2)
\]

value of \(x\): 1

At first glance: \(x\) is assigned 3
But: both parallel components could read \(x\) before it is written
Thus: \(x\) is assigned 2, 1,
Concurrency and Interaction by Example

Observation: concurrency introduces new phenomena

Example 1.1

\[ x := 0; \\
(x := x + 1 \parallel x := x + 2) \quad \text{value of } x: 0 \]

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
- Thus: \( x \) is assigned 2, 1,
Observation: concurrency introduces new phenomena

Example 1.1

\[
\begin{align*}
x & := 0; \\
(x := x + 1) \parallel (x := x + 2) & \quad \text{value of } x: 0 \\
& \quad 2
\end{align*}
\]

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
- Thus: \( x \) is assigned 2, 1,
Observation: concurrency introduces new phenomena

Example 1.1

\[
\begin{align*}
x & := 0; \\
( & x := x + 1 \parallel x := x + 2) \quad \text{value of } x: 2 \\
& 2
\end{align*}
\]

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
- Thus: \( x \) is assigned 2, 1,
Observation: concurrency introduces new phenomena

Example 1.1

\[
x := 0; \\
(x := x + 1 \parallel x := x + 2)
\]

value of \(x\): 2

- At first glance: \(x\) is assigned 3
- But: both parallel components could read \(x\) before it is written
- Thus: \(x\) is assigned 2, 1,
Observation: concurrency introduces new phenomena

Example 1.1

\[ x := 0; \]
\[(x := x + 1 \parallel x := x + 2) \quad \text{value of } x: 3 \]

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
- Thus: \( x \) is assigned 2, 1, or 3
Concurrent and Interaction by Example

Observation: concurrency introduces new phenomena

Example 1.1

\[
\begin{align*}
x &:= 0; \\
(x &:= x + 1 \parallel x := x + 2)
\end{align*}
\]

- At first glance: \( x \) is assigned 3
- But: both parallel components could read \( x \) before it is written
- Thus: \( x \) is assigned 2, 1, or 3
- If exclusive access to shared memory and atomic execution of assignments guaranteed
  \( \Rightarrow \) only possible outcome: 3
Concurrency and Interaction

The problem arises due to the combination of

- concurrency and
- interaction (here: via shared memory)
Concurrency and Interaction

The problem arises due to the combination of

- concurrency and
- interaction (here: via shared memory)

Conclusion

When modelling concurrent systems, the precise description of the mechanisms of both concurrency and interaction is crucially important.
Concurrency Everywhere

Herb Sutter: *The Free Lunch Is Over*, Dr. Dobb’s Journal, 30(3), 2005

“The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency.”

- Operating systems
- Embedded/reactive systems
  - parallelism (at least) between hardware, software, and environment
- High-end parallel hardware infrastructure:
  - high-performance computing
- Low-end parallel hardware infrastructure
  - increasing performance only achievable by parallelism
  - multi-core computers, GPGPUs, FPGAs

*Moore’s Law:* Transistor density doubles every 2 years
Problems Everywhere

- Operating systems:
  - mutual exclusion
  - fairness (no starvation)
  - no deadlocks, ...
- Shared-memory systems:
  - memory models
  - data races
  - inconsistencies
    (“sequential consistency” vs. relaxed notions)
- Embedded systems:
  - safety
  - liveness, ...
Outline of Lecture 1

Preliminaries

Concurrency and Interaction

A Closer Look at Memory Models

A Closer Look at Reactive Systems

Overview of the Course

The Approach

Syntax of CCS

Intuitive Meaning and Examples

Formal Semantics of CCS

Infinite State Spaces

The CAAL Tool
Memory Models

An illustrative example

Initially: \( x = y = 0 \)

**thread1:**

1: \( x = 1 \)

2: \( r1 = y \)

**thread2:**

3: \( y = 1 \)

4: \( r2 = x \)

(with global variables \( x, y \) and local registers \( r1, r2 \) )
Memory Models

Sequential Consistency (SC)

<table>
<thead>
<tr>
<th>T1</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=1</td>
<td>x = 0</td>
</tr>
<tr>
<td>r1=y</td>
<td>y = 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>y=1</td>
</tr>
<tr>
<td>r2=x</td>
</tr>
</tbody>
</table>
Sequential Consistency (SC)

T1

x = 1

r1 = y

T2

y = 1

r2 = x

Memory Model:

- r1 = y
  - y = 1
- r2 = x
  - x = 0
  - y = 0
Memory Models

Sequential Consistency (SC)

Memory
- x = 1
- y = 0

T1
- r1 = y
- x = 1

T2
- r2 = x
- y = 1
Memory Models

Sequential Consistency (SC)

Memory

T1

T2

x = 1
y = 0

x = 1
y = 0

r1 = y

r2 = x

r1 = y

y = 1
Memory Models

Sequential Consistency (SC)

Memory

\[ x = 1 \]

\[ y = 0 \]

T1

\[ x = 1 \]

\[ r1 = y \]

[r1 = 0]

T2

\[ y = 1 \]

\[ r2 = x \]

Tuesday, April 5, 2011
Sequential Consistency (SC)

T1

x = 1

r1 = y

T2

y = 1

r2 = x

Memory

x = 1

y = 0

[r1 = 0]
Sequential Consistency (SC)

x = 1
y = 1

x = 1
y = 1

r1 = y
[r1 = 0]

r2 = x
y = 1

Memory Models
Sequential Consistency (SC)

Memory

\[ x = 1 \]
\[ y = 1 \]

T1

- \( x = 1 \)
- \( r_1 = y \) [\( r_1 = 0 \)]

T2

- \( y = 1 \)
- \( r_2 = x \)
Memory Models

Sequential Consistency (SC)

T1

\[ \begin{align*}
    x &= 1 \\
    r1 &= y \\
    r1 &= 0
\end{align*} \]

T2

\[ \begin{align*}
    y &= 1 \\
    r2 &= x \\
    r2 &= 1
\end{align*} \]
**Memory Models**

Sequential Consistency (SC)

- **Memory**
  - $x = 1$
  - $y = 1$

- **T1**
  - $x = 1$
  - $r1 = y$ [r1=0]

- **T2**
  - $y = 1$
  - $r2 = x$ [r2=1]

- **Not** $(r1 == 0 \text{ and } r2 == 0)$
Memory Models

Total Store Ordering (TSO)

T1

x = 1
r1 = y

T2

y = 1
r2 = x

Memory

x = 0
y = 0

Tuesday, April 5, 2011
Memory Models

Total Store Ordering (TSO)

FIFO buffer T1

T1

x = 1

r1 = y

T2

y = 1

r2 = x
Memory Models

Total Store Ordering (TSO)

- FIFO buffer T1
  - T1
    - x = 1
    - r1 = y

- Memory
  - x = 0
  - y = 0

- FIFO buffer T2
  - T2
    - y = 1
    - r2 = x
Memory Models

Total Store Ordering (TSO)

FIFO buffer T1

T1

x=1

r1=y

Memory

x = 0
y = 0

T2

y=1

r2=x

FIFO buffer T2
Memory Models

Total Store Ordering (TSO)

- FIFO buffer T1
- Memory: x = 0, y = 0
- FIFO buffer T2

T1
- x = 1
- r1 = y
- [r1 = 0]

T2
- y = 1
- r2 = x
Memory Models

Total Store Ordering (TSO)

FIFO buffer T1

T1

x = 1

r1 = y

y = 1

r2 = x

x = 0

y = 0

FIFO buffer T2

T2

y = 1

r2 = x
Memory Models

Total Store Ordering (TSO)

T1

FIFO buffer T1

x=1

T2

FIFO buffer T2

x=0

y=0

T

r1=y

Memory

[r1=0]

r2=x
Memory Models

Total Store Ordering (TSO)

FIFO buffer T1

T1

x = 1

r1 = y

y = 1

[y = 0]

Memory

x = 1

r1 = 0

y = 1

r2 = x

[r1 = 0]

[r2 = 0]

FIFO buffer T2

T2

y = 1

r2 = x

[r2 = 0]

r1 == 0 and r2 == 0
Outline of Lecture 1

Preliminaries

Concurrency and Interaction

A Closer Look at Memory Models

A Closer Look at Reactive Systems

Overview of the Course

The Approach

Syntax of CCS

Intuitive Meaning and Examples

Formal Semantics of CCS

Infinite State Spaces

The CAAL Tool
Reactive Systems I

- “Classical” model for sequential systems
  
  \[ \text{System : Input} \rightarrow \text{Output} \]
  
  (transformational systems) is not adequate

- Missing: aspect of interaction
Reactive Systems I

- “Classical” model for sequential systems

  \[ \text{System} : \text{Input} \rightarrow \text{Output} \]

  (transformational systems) is not adequate

- Missing: aspect of interaction

- Rather: reactive systems which interact with environment and among themselves
Reactive Systems I

- “Classical” model for sequential systems

\[ \text{System : Input} \rightarrow \text{Output} \]

(transformational systems) is not adequate

- Missing: aspect of interaction

- Rather: reactive systems which interact with environment and among themselves

- Main interest: not terminating computations but infinite behaviour (system maintains ongoing interaction with environment)

- Examples:
  - operating systems
  - embedded systems controlling mechanical or electrical devices (planes, cars, home appliances, ...)
  - power plants, production lines, ...
Observation

Reactive systems are often **safety critical**, thus **trustworthiness** has to be ensured.

- **Safety** properties: “Nothing bad is ever going to happen.”
  - e.g., “at most one process in the critical section”
- **Liveness** properties: “Eventually something good will happen.”
  - e.g., “every request will finally be answered by the server”
- **Fairness** properties: “No component will starve to death.”
  - e.g., “any process requiring entry to the critical section will eventually be admitted”
- Reliability, performance, survivability, ...
Outline of Lecture 1

Preliminaries

Concurrency and Interaction

A Closer Look at Memory Models

A Closer Look at Reactive Systems

Overview of the Course

The Approach

Syntax of CCS

Intuitive Meaning and Examples

Formal Semantics of CCS

Infinite State Spaces

The CAAL Tool
## Overview of the Course

<table>
<thead>
<tr>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(1) Milner’s Calculus of Communicating Systems (CCS)</strong></td>
</tr>
<tr>
<td>– introduction and motivation</td>
</tr>
<tr>
<td>– syntax of CCS</td>
</tr>
<tr>
<td>– semantics of CCS</td>
</tr>
<tr>
<td>– the CAAL tool</td>
</tr>
<tr>
<td><strong>(2) Behavioural Equivalences</strong></td>
</tr>
<tr>
<td>– trace equivalence</td>
</tr>
<tr>
<td>– bisimulation</td>
</tr>
<tr>
<td>– congruence</td>
</tr>
<tr>
<td>– deadlock sensitivity</td>
</tr>
<tr>
<td><strong>(3) Logical Specifications</strong></td>
</tr>
<tr>
<td>– Hennessy-Milner Logic</td>
</tr>
<tr>
<td>– HML and traces</td>
</tr>
<tr>
<td>– HML and bisimulation</td>
</tr>
<tr>
<td>– adding recursion</td>
</tr>
<tr>
<td><strong>(4) Application: Mutual-Exclusion Protocols</strong></td>
</tr>
<tr>
<td>– modelling mutex algorithms in CCS</td>
</tr>
<tr>
<td>– verification by model checking</td>
</tr>
<tr>
<td>– verification by bisimulation checking</td>
</tr>
</tbody>
</table>
## Overview of the Course

### Overview

1. **Milner’s Calculus of Communicating Systems (CCS)**
   - introduction and motivation
   - syntax of CCS
   - semantics of CCS
   - the CAAL tool
2. **Behavioural Equivalences**
   - trace equivalence
   - bisimulation
   - congruence
   - deadlock sensitivity
3. **Logical Specifications**
   - Hennessy-Milner Logic
   - HML and traces
   - HML and bisimulation
   - adding recursion
4. **Application: Mutual-Exclusion Protocols**
   - modelling mutex algorithms in CCS
   - verification by model checking
   - verification by bisimulation checking

### Literature

Outline of Lecture 1

Preliminaries
Concurrency and Interaction
A Closer Look at Memory Models
A Closer Look at Reactive Systems
Overview of the Course

The Approach

Syntax of CCS
Intuitive Meaning and Examples
Formal Semantics of CCS
Infinite State Spaces
The CAAL Tool
# The Calculus of Communicating Systems

## History

- **First development:**

- **Elaboration and larger case studies:**

- **Extension to mobile systems:**
The Calculus of Communicating Systems

History

- First development:
- Elaboration and larger case studies:
- Extension to mobile systems:
  Robin Milner: *Communicating and Mobile Systems: the \( \pi \)-calculus*, Cambridge University Press, 1999

Approach

Description of concurrency on a **simple and abstract level**, using only a few basic primitives

- no explicit storage (variables)
- no explicit representation of values (numbers, Booleans, ...), or data structures

\[\Rightarrow\] Concurrent system reduced to **communication potential**
Outline of Lecture 1

Preliminaries
Concurrency and Interaction
A Closer Look at Memory Models
A Closer Look at Reactive Systems
Overview of the Course
The Approach

Syntax of CCS
Intuitive Meaning and Examples
Formal Semantics of CCS
Infinite State Spaces
The CAAL Tool
Definition 1.2 (Syntax of CCS)

- Let $A$ be a set of (action) names.
Definition 1.2 (Syntax of CCS)

- Let $A$ be a set of (action) names.
- $\bar{A} := \{\bar{a} \mid a \in A\}$ denotes the set of co-names.
Syntax of CCS I

Definition 1.2 (Syntax of CCS)

- Let $A$ be a set of (action) names.
- $\overline{A} := \{\overline{a} \mid a \in A\}$ denotes the set of co-names.
- $Act := A \cup \overline{A} \cup \{\tau\}$ is the set of actions with the silent (or: unobservable) action $\tau$. 
Definition 1.2 (Syntax of CCS)

- Let \( A \) be a set of (action) names.
- \( \overline{A} := \{ \overline{a} \mid a \in A \} \) denotes the set of co-names.
- \( Act := A \cup \overline{A} \cup \{ \tau \} \) is the set of actions with the silent (or: unobservable) action \( \tau \).
- Let \( Pid \) be a set of process identifiers.
### Definition 1.2 (Syntax of CCS)

- Let $A$ be a set of (action) names.
- $\overline{A} := \{ \overline{a} \mid a \in A \}$ denotes the set of co-names.
- $\text{Act} := A \cup \overline{A} \cup \{ \tau \}$ is the set of actions with the silent (or: unobservable) action $\tau$.
- Let $\text{Pid}$ be a set of process identifiers.
- The set $\text{Prc}$ of process expressions is defined by the following syntax:

$$ P ::= \text{nil} \quad \text{(inaction)} $$

$$ \quad \mid \alpha.P \quad \text{(prefixing)} $$

$$ \quad \mid P_1 + P_2 \quad \text{(choice)} $$

$$ \quad \mid P_1 \parallel P_2 \quad \text{(parallel composition)} $$

$$ \quad \mid P \setminus L \quad \text{(restriction)} $$

$$ \quad \mid P[f] \quad \text{(relabelling)} $$

$$ \quad \mid C \quad \text{(process call)} $$

where $\alpha \in \text{Act}$, $\emptyset \neq L \subseteq A$, $C \in \text{Pid}$, and $f : \text{Act} \to \text{Act}$ such that $f(\tau) = \tau$ and $f(a) = f(a)$ for each $a \in A$. 

---

21 of 37 Modelling and Analysing Concurrent Systems
RIO 2023
Lecture 1: Milner’s Calculus of Communicating Systems
Definition 1.2 (continued)

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

\[(C_i = P_i \mid 1 \leq i \leq k)\]

where \(k \geq 1\), \(C_i \in Pid\) (pairwise distinct), and \(P_i \in Prc\) (with identifiers from \(\{C_1, \ldots, C_k\}\)).
Definition 1.2 (continued)

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

\[(C_i = P_i \mid 1 \leq i \leq k)\]

where \(k \geq 1\), \(C_i \in Pid\) (pairwise distinct), and \(P_i \in Prc\) (with identifiers from \(\{C_1, \ldots, C_k\}\)).

Notational Conventions:

- \(\overline{a}\) means \(a\)
- \(\sum_{i=1}^{n} P_i (n \in \mathbb{N})\) means \(P_1 + \ldots + P_n\) (where \(\sum_{i=1}^{0} P_i := \text{nil}\))
- \(P \setminus a\) abbreviates \(P \setminus \{a\}\)
- \([a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]\) stands for \(f : Act \rightarrow Act\) with \(f(a_i) = b_i\) for \(i \in [n]\) and \(f(\alpha) = \alpha\) otherwise
- Restriction and relabelling bind stronger than prefixing, prefixing stronger than composition, composition stronger than choice:

\[P \setminus a + b.Q \parallel R\] means \((P \setminus a) + ((b.Q) \parallel R)\)
Outline of Lecture 1

Preliminaries
Concurrency and Interaction
A Closer Look at Memory Models
A Closer Look at Reactive Systems
Overview of the Course
The Approach
Syntax of CCS

**Intuitive Meaning and Examples**

Formal Semantics of CCS
Infinite State Spaces
The CAAL Tool
Meaning of CCS Constructs

- **nil** is an inactive process that can do nothing.
Meaning of CCS Constructs

- **nil** is an inactive process that can do nothing.
- $\alpha.P$ can execute $\alpha$ and then behaves as $P$. 
Meaning of CCS Constructs

- nil is an inactive process that can do nothing.
- $\alpha.P$ can execute $\alpha$ and then behaves as $P$.
- An action $a \in A$ ($\bar{a} \in \bar{A}$) is interpreted as an input (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in $P_1 \parallel P_2$), they are merged into a $\tau$-action.
Meaning of CCS Constructs

- nil is an inactive process that can do nothing.
- $\alpha.P$ can execute $\alpha$ and then behaves as $P$.
- An action $a \in A$ ($\bar{a} \in \bar{A}$) is interpreted as an input (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in $P_1 \parallel P_2$), they are merged into a $\tau$-action.
- $P_1 + P_2$ represents the nondeterministic choice between $P_1$ and $P_2$. 
Meaning of CCS Constructs

- nil is an inactive process that can do nothing.
- $\alpha.P$ can execute $\alpha$ and then behaves as $P$.
- An action $a \in A$ ($\bar{a} \in \bar{A}$) is interpreted as an input (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in $P_1 \parallel P_2$), they are merged into a $\tau$-action.
- $P_1 + P_2$ represents the nondeterministic choice between $P_1$ and $P_2$.
- $P_1 \parallel P_2$ denotes the parallel execution of $P_1$ and $P_2$, involving interleaving or communication.
Meaning of CCS Constructs

- nil is an inactive process that can do nothing.
- \( \alpha.P \) can execute \( \alpha \) and then behaves as \( P \).
- An action \( a \in A \ (\bar{a} \in \bar{A}) \) is interpreted as an input (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in \( P_1 \parallel P_2 \)), they are merged into a \( \tau \)-action.
- \( P_1 + P_2 \) represents the nondeterministic choice between \( P_1 \) and \( P_2 \).
- \( P_1 \parallel P_2 \) denotes the parallel execution of \( P_1 \) and \( P_2 \), involving interleaving or communication.
- The restriction \( P \setminus L \) declares each \( a \in L \) as a local name which is only known within \( P \).
Meaning of CCS Constructs

- **nil** is an inactive process that can do nothing.
- **α.P** can execute α and then behaves as P.
- An action \( a \in A (\bar{a} \in \bar{A}) \) is interpreted as an input (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in \( P_1 \parallel P_2 \)), they are merged into a \( \tau \)-action.
- **\( P_1 + P_2 \)** represents the nondeterministic choice between \( P_1 \) and \( P_2 \).
- **\( P_1 \parallel P_2 \)** denotes the parallel execution of \( P_1 \) and \( P_2 \), involving interleaving or communication.
- The restriction **\( P \setminus L \)** declares each \( a \in L \) as a local name which is only known within \( P \).
- The relabelling **\( P[f] \)** allows to adapt the naming of actions.
Meaning of CCS Constructs

- **nil** is an *inactive process* that can do nothing.
- **\( \alpha . P \)** can execute \( \alpha \) and then behaves as \( P \).
- An action \( a \in A \ (\bar{a} \in \bar{A}) \) is interpreted as an *input* (output, resp.) operation. Both are complementary: if performed in parallel (i.e., in \( P_1 \parallel P_2 \)), they are merged into a \( \tau \)-action.
- \( P_1 + P_2 \) represents the *nondeterministic choice* between \( P_1 \) and \( P_2 \).
- \( P_1 \parallel P_2 \) denotes the *parallel execution* of \( P_1 \) and \( P_2 \), involving *interleaving* or communication.
- The *restriction* \( P \setminus L \) declares each \( a \in L \) as a local name which is only known within \( P \).
- The *relabelling* \( P[f] \) allows to adapt the naming of actions.
- The behaviour of a *process call* \( C \) is given by the right-hand side of the corresponding equation.
Example 1.3

(1) One-place buffer:

\[ B = \text{in} \cdot \text{out} \cdot B \]
Example 1.3

(1) One-place buffer:

\[ B = \text{in} \cdot \text{out} \cdot B \]

(2) Two-place buffer:

\[ B_0 = \text{in} \cdot B_1 \]
\[ B_1 = \text{out} \cdot B_0 + \text{in} \cdot B_2 \]
\[ B_2 = \text{out} \cdot B_1 \]
Example 1.3

(1) One-place buffer:

\[ B = \text{in} \cdot \text{out} \cdot B \]

(2) Two-place buffer:

\[ B_0 = \text{in} \cdot B_1 \]
\[ B_1 = \text{out} \cdot B_0 + \text{in} \cdot B_2 \]
\[ B_2 = \text{out} \cdot B_1 \]

(3) Parallel two-place buffer:

\[ B_\parallel = (B_{\text{out} \leftrightarrow \text{com}} \parallel B_{\text{in} \leftrightarrow \text{com}}) \setminus \text{com} \]
\[ B = \text{in} \cdot \text{out} \cdot B \]

“Interaction diagram”:
Outline of Lecture 1

Preliminaries
Concurrency and Interaction
A Closer Look at Memory Models
A Closer Look at Reactive Systems
Overview of the Course
The Approach
Syntax of CCS
Intuitive Meaning and Examples

Formal Semantics of CCS
Infinite State Spaces
The CAAL Tool
Labelled Transition Systems

**Goal:** represent system behaviour by (infinite) graph
- nodes = system states
- edges = transitions between states
Labelled Transition Systems

**Goal:** represent system behaviour by (infinite) graph
- nodes = system states
- edges = transitions between states

**Definition 1.4 (Labelled transition system)**

A labelled transition system (LTS) is a triple \((S, Act, \rightarrow)\) consisting of
- a set \(S\) of states
- a set \(Act\) of (action) labels
- a transition relation \(\rightarrow \subseteq S \times Act \times S\)

For \((s, \alpha, s') \in \rightarrow\) we write \(s \xrightarrow{\alpha} s'\). An LTS is called finite if \(S\) is so.
Labelled Transition Systems

Goal: represent system behaviour by (infinite) graph
- nodes = system states
- edges = transitions between states

Definition 1.4 (Labelled transition system)

A labelled transition system (LTS) is a triple \((S, Act, \rightarrow)\) consisting of
- a set \(S\) of states
- a set \(Act\) of (action) labels
- a transition relation \(\rightarrow \subseteq S \times Act \times S\)

For \((s, \alpha, s') \in \rightarrow\) 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 ("LTS\(s_0\)").
- (Finite) LTSs correspond to (finite) automata without final states.
We define the assignment

\[ \text{syntax} \rightarrow \text{semantics} \]

\[ \text{process definition} \leftrightarrow \text{LTS} \]

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

\[ \text{premise(s)} \]

\[ \text{(rule name)} \]

\[ \text{conclusion} \]

which are composed to form derivation trees (where axioms, i.e., rules without premises, correspond to leaves).
Semantics of CCS II

Reminder:  \( P ::= \text{nil} \mid \alpha.P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid P \setminus L \mid P[f] \mid C \)

Definition 1.5 (Semantics of CCS)

A process definition \((C_i = P_i \mid 1 \leq i \leq k)\) determines the LTS \((Prc, Act, \rightarrow)\) whose transitions can be inferred from the following rules \((P, P', Q, Q' \in Prc, \alpha \in Act, \lambda \in A \cup \bar{A}, a \in A)\):

\[
\begin{align*}
\text{(Act)} & : \quad \alpha . P \xrightarrow{\alpha} P \\
\text{(Sum1)} & : \quad P \xrightarrow{\alpha} P' \quad P + Q \xrightarrow{\alpha} P' \\
\text{(Sum2)} & : \quad Q \xrightarrow{\alpha} Q' \quad P + Q \xrightarrow{\alpha} Q' \\
\text{(Par1)} & : \quad P \xrightarrow{\alpha} P' \quad P \parallel Q \xrightarrow{\alpha} P' \parallel Q \\
\text{(Par2)} & : \quad Q \xrightarrow{\alpha} Q' \quad P \parallel Q \xrightarrow{\alpha} P \parallel Q' \\
\text{(Com)} & : \quad P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\bar{\lambda}} Q' \\
\text{(Res)} & : \quad P \xrightarrow{\alpha} P' \quad (\alpha, \bar{\alpha} \notin L) \\
\text{(Rel)} & : \quad P \xrightarrow{\alpha} P' \quad P[f] \xrightarrow{f(\alpha)} P'[f] \\
\text{(Call)} & : \quad C \xrightarrow{\alpha} P' \quad (C = P)
\end{align*}
\]
Example 1.6

(1) One-place buffer: $B = \text{in}\overline{\text{out}}.B$

- First step:

\[
\begin{align*}
\text{(Act)} & \quad \overline{\text{in}\overline{\text{out}}.B} \xrightarrow{\text{in}} \overline{\text{out}}.B \\
\text{(Call)} & \quad \overline{B} \xrightarrow{\text{in}} \overline{\text{out}}.B
\end{align*}
\]
Example 1.6

(1) One-place buffer: $B = \text{in.out}.B$

- First step:

  \[
  \begin{align*}
  &\text{(Call)} \quad \text{in.out}.B \xrightarrow{\text{in}} \text{out}.B \\
  &\text{(Act)} \quad B \xrightarrow{\text{in}} \text{out}.B
  \end{align*}
  \]

- Second step:

  \[
  \begin{align*}
  &\text{(Act)} \quad \text{out}.B \xrightarrow{\text{out}} B
  \end{align*}
  \]
(1) One-place buffer: $B = \text{in}.\overline{\text{out}}.B$

- First step:

\[
\text{(Act)} \quad \text{in}.\overline{\text{out}}.B \xrightarrow{\text{in}} \overline{\text{out}}.B \\
\text{(Call)} \quad B \xrightarrow{\text{in}} \overline{\text{out}}.B
\]

- Second step:

\[
\text{(Act)} \quad \overline{\text{out}}.B \xrightarrow{\text{out}} B
\]

⇒ Complete LTS:
Example 1.6 (continued)

(2) Sequential two-place buffer: $B_0 = \text{in}.B_1$
    $B_1 = \text{out}.B_0 + \text{in}.B_2$
    $B_2 = \text{out}.B_1$

- First step:

  \[
  \begin{array}{c}
  \text{(Act)} \quad \text{in}.B_1 \xrightarrow{\text{in}} B_1 \\
  \text{(Call)} \quad B_0 \xrightarrow{\text{in}} B_1
  \end{array}
  \]
Example 1.6 (continued)

(2) Sequential two-place buffer:

\[ \begin{align*}
B_0 &= \text{in}.B_1 \\
B_1 &= \overline{\text{out}}.B_0 + \text{in}.B_2 \\
B_2 &= \overline{\text{out}}.B_1
\end{align*} \]

- First step:

\[ \begin{align*}
\text{(Act)} & \quad \text{in}.B_1 \xrightarrow{\text{in}} B_1 \\
\text{(Call)} & \quad B_0 \xrightarrow{\text{in}} B_1
\end{align*} \]

- Second step:

\[ \begin{align*}
\text{(Act)} & \quad \overline{\text{out}}.B_0 \xrightarrow{\text{out}} B_0 \\
\text{(Sum_1)} & \quad \overline{\text{out}}.B_0 + \text{in}.B_2 \xrightarrow{\text{out}} B_0 \\
\text{(Call)} & \quad B_1 \xrightarrow{\overline{\text{out}}} B_0
\end{align*} \]
Example 1.6 (continued)

(2) Sequential two-place buffer:

\[ B_0 = \text{in}.B_1 \]
\[ B_1 = \text{out}.B_0 + \text{in}.B_2 \]
\[ B_2 = \text{out}.B_1 \]

- First step:

\[
\begin{array}{c}
\text{(Act)} \quad \text{in}.B_1 \xrightarrow{\text{in}} B_1 \\
\text{(Call)} \quad B_0 \xrightarrow{\text{in}} B_1
\end{array}
\]

- Like second step (with (Sum$_2$)): \[ B_1 \xrightarrow{\text{in}} B_2 \]

- Like first step: \[ B_2 \xrightarrow{\text{out}} B_1 \]

- Second step:

\[
\begin{array}{c}
\text{(Act)} \quad \text{out}.B_0 \xrightarrow{\text{out}} B_0 \\
\text{(Sum$_1$)} \quad \text{out}.B_0 + \text{in}.B_2 \xrightarrow{\text{out}} B_0
\end{array}
\]

\[
\begin{array}{c}
\text{(Call)} \quad B_1 \xrightarrow{\text{out}} B_0
\end{array}
\]
Example 1.6 (continued)

(2) Sequential two-place buffer: \( B_0 = in.B_1 \)
\( B_1 = out.B_0 + in.B_2 \)
\( B_2 = out.B_1 \)

- First step:
  \[
  \begin{array}{c}
  \text{(Act)} \quad in.B_1 \xrightarrow{in} B_1 \\
  \text{(Call)} \quad B_0 \xrightarrow{in} B_1
  \end{array}
  \]

- Second step:
  \[
  \begin{array}{c}
  \text{(Act)} \quad out.B_0 \xrightarrow{out} B_0 \\
  \text{(Sum}_1 \text{)} \quad out.B_0 + in.B_2 \xrightarrow{out} B_0 \\
  \text{(Call)} \quad B_1 \xrightarrow{out} B_0
  \end{array}
  \]

- Like second step (with \( \text{(Sum}_2 \text{)} \)): \( B_1 \xrightarrow{in} B_2 \)

- Like first step: \( B_2 \xrightarrow{out} B_1 \)

- Complete LTS:
  \[
  \begin{array}{c}
  \xrightarrow{in} \quad B_0 \xrightarrow{\text{empty}} B_1 \xrightarrow{\text{one entry}} B_2 \xrightarrow{\text{full}} \xrightarrow{\text{out}} \xrightarrow{\text{out}} B_0
  \end{array}
  \]
Example 1.6 (continued)

(3) Parallel two-place buffer:

\[ B \parallel = (B[f] \parallel B[g]) \setminus \text{com} \]
\[ B = \text{in.out.B} \]

\((f := [\text{out} \leftrightarrow \text{com}], \ g := [\text{in} \leftrightarrow \text{com}])\)

First step:

\[ \begin{align*}
\text{(Act)} & \quad \frac{\text{in.out.B} \xrightarrow{\text{in}} \text{out.B}}{
\text{(Call)} & \quad \frac{B \xrightarrow{\text{in}} \text{out.B}}{
\text{(Rel)} & \quad \frac{B[f] \xrightarrow{\text{in}} (\overline{\text{out.B}})[f]}{
\text{(Par)} & \quad \frac{B[f] \parallel B[g] \xrightarrow{\text{in}} (\overline{\text{out.B}})[f] \parallel B[g]}{
\text{(Res)} & \quad \frac{(B[f] \parallel B[g]) \setminus \text{com} \xrightarrow{\text{in}} ((\overline{\text{out.B}})[f] \parallel B[g]) \setminus \text{com}}{
\text{(Call)} & \quad \frac{B \parallel \xrightarrow{\text{in}} ((\overline{\text{out.B}})[f] \parallel B[g]) \setminus \text{com}}
\end{align*} \]
Example 1.6 (continued)

(3) Parallel two-place buffer: \( B \parallel = (B[f] \parallel B[g]) \setminus \text{com} \)  
\( f := [\text{out} \leftrightarrow \text{com}], \ g := [\text{in} \leftrightarrow \text{com}] \)  
\( B = \text{in}.\text{out}.B \)

Complete LTS:

\[
\begin{array}{c}
\xrightarrow{\text{in}} \quad B \parallel \\
\xrightarrow{\text{in}} \\
\xrightarrow{\text{out}} \\
\xrightarrow{\tau} \\
\xrightarrow{\text{out}} \\
\xrightarrow{\text{in}} \\
\end{array}
\quad \begin{array}{c}
(B[f] \parallel B[g]) \setminus \text{com} \\
(B[f] \parallel (\text{out}.B)[g]) \setminus \text{com} \\
((\text{out}.B)[f] \parallel (\text{out}.B)[g]) \setminus \text{com} \\
\text{empty} \\
\text{one entry} \\
\text{full} \\
\end{array}
\]
Outline of Lecture 1

Preliminaries
Concurrency and Interaction
A Closer Look at Memory Models
A Closer Look at Reactive Systems
Overview of the Course
The Approach
Syntax of CCS
Intuitive Meaning and Examples
Formal Semantics of CCS

Infinite State Spaces
The CAAL Tool
The Power of Recursive Definitions

So far: only finite state spaces – not necessarily true!
The Power of Recursive Definitions

So far: only finite state spaces – not necessarily true!

Example 1.7 (Counter)

\[ C = up.(C \parallel down.nil) \]
The Power of Recursive Definitions

So far: only finite state spaces – not necessarily true!

Example 1.7 (Counter)

\[
C = up.(C \parallel \text{down.nil})
\]

gives rise to infinite LTS (abbreviating \(\text{down} := \text{down.nil}\)):
The Power of Recursive Definitions

So far: only finite state spaces – not necessarily true!

Example 1.7 (Counter)

\[ C = up.(C \parallel down.nil) \]

gives rise to infinite LTS (abbreviating \( down := down.nil \)):

Sequential “specification”:

\[ C_0 = up.C_1 \]
\[ C_n = up.C_{n+1} + down.C_{n-1} \quad (n > 0) \]
Outline of Lecture 1

Preliminaries
Concurrency and Interaction
A Closer Look at Memory Models
A Closer Look at Reactive Systems
Overview of the Course
The Approach
Syntax of CCS
Intuitive Meaning and Examples
Formal Semantics of CCS
Infinite State Spaces
The CAAL Tool
The **CAAL Tool**

**CAAL (Concurrency Workbench, Aalborg Edition; https://caal.cs.aau.dk/)**

- Smart editor
- Visualisation of generated LTS
- Equivalence checking w.r.t. several bisimulation, simulation and trace equivalences
- Generation of distinguishing formulae for non-equivalent processes
- Model checking of recursive HML formulae
- (Bi)simulation and model checking games.