Languages for Concurrency and Distribution Simple (for real)

Gabriel Rovesti

April 19, 2025

Contents

1 Introduction to the Course

Concurrent computation is a form of computation in which several computations are executed during overlapping time periods, rather than sequentially. This course studies both the theoretical foundations of concurrency and its practical aspects in modern programming languages.

1.1 Key Concepts in Concurrency

Concurrency theory provides mathematical frameworks for modeling concurrent systems:

- **Process**: A sequential program in execution
- **Concurrent System**: Multiple processes that may interact with each other
- **Communication**: How processes exchange information
- **Synchronization**: Coordination of the execution of processes

1.2 Challenges in Concurrent Programming

Concurrent programming introduces several challenges not present in sequential programming:

- **Race Conditions**: When the outcome depends on the timing of uncontrollable events
- **Deadlocks**: When two or more processes are unable to proceed because each is waiting for resources held by the other
- **Livelocks**: When processes continuously change their states without making progress
- **Resource Starvation**: When a process is perpetually denied necessary resources

1.3 Models of Concurrency

Several mathematical models have been developed to describe concurrent systems:

- **Process Calculi**: CCS, -calculus
- **Petri Nets**: Mathematical modeling language for distributed systems
- **Automata Theory**: Extended for concurrent systems
- **Temporal Logic**: For specifying and verifying properties of concurrent systems

1.4 Course Overview

This course is structured into two main parts:

1. **Theoretical Foundations**: Process calculi, bisimulation, fixed point theory, and modal logics
2. **Practical Aspects**: Programming languages with built-in concurrency features (Go, Erlang, Clojure)

2 The Language CCS, Informally

Calculus of Communicating Systems (CCS) is a process calculus for modeling concurrent systems, developed by Robin Milner in the late 1970s.

2.1 Basic Intuition

CCS models a system as a collection of processes that interact through synchronized communication. Each process performs actions and can evolve into other processes. Communication occurs when complementary actions (send and receive) on the same channel are performed by different processes.

2.2 Core Concepts

- **Actions**:
  - Input actions (denoted as $a, b, c, \ldots$)
  - Output actions (denoted as $\overline{a}, \overline{b}, \overline{c}, \ldots$)
  - Silent or internal action (denoted as τ)
- **Operators**:
  - Prefix ($a.P$): Perform action a and then behave as process P
  - Summation ($P + Q$): Behave either as P or as Q
  - Parallel composition ($P \mid Q$): P and Q execute concurrently
  - Restriction ($P \setminus L$): Restrict actions in set L from being observable
  - Relabeling ($P[f]$): Rename actions according to function f
  - Recursion (rec $X.P$): Define recursive behavior

2.3 Communication and Synchronization

Communication in CCS occurs when two processes perform complementary actions. For example, if process P can perform an input action a and process Q can perform the corresponding output action $\overline{a}$, then their parallel composition $P \mid Q$ can perform a τ action, representing internal communication.

2.4 Example: Simple Buffer

A one-place buffer that can receive an input and then produce it as output can be represented as:

$$\text{Buffer} = \text{in}.(\text{out}.\text{Buffer})$$

This process first performs an input action (in), then an output action (out), and then behaves as Buffer again.

3 The Language CCS: Formal Syntax and Operational Semantics

3.1 Formal Syntax

The syntax of CCS can be formally defined using BNF (Backus-Naur Form). We first define the basic elements:

- A countably infinite set of channel names: $\mathcal{A} = \{a, b, c, \ldots\}$
- A set of actions: $\text{Act} = \mathcal{A} \cup \{\overline{a} \mid a \in \mathcal{A}\} \cup \{\tau\}$, where:
  - $a \in \mathcal{A}$ represents an input action on channel a
  - $\overline{a}$ represents an output action on channel a
  - τ represents an internal action
- A set of process constants: $\mathcal{K} = \{K, K', K_1, K_2, \ldots\}$

The CCS processes are defined by:
$P ::= 0 \mid \alpha.P \mid P + P \mid P \mid P \mid P \setminus L \mid P[f] \mid K \mid \text{rec } X.P$
Where:

- 0 is the inactive process, sometimes identified with empty sum $\sum_{i \in \emptyset} P_i$
- $\alpha \in \text{Act}$ is an action
- $P + P$ represents non-deterministic choice
- $P \mid P$ represents parallel composition
- $P \setminus L$ is the restriction operator, where $L \subseteq \mathcal{A}$
- $P[f]$ is the relabeling operator, where $f : \text{Act} \to \text{Act}$ is a function such that $f(\tau) = \tau$ and $f(\overline{a}) = \overline{f(a)}$
- $K \in \mathcal{K}$ is a process constant, defined by an equation $K \stackrel{\text{def}}{=} P$
- rec $X.P$ represents recursion, where X is a process variable

3.2 Operational Semantics

The operational semantics of CCS is defined by a labeled transition system (LTS), where transitions between process states are labeled with actions. The semantics is given by syntax-driven inference rules in the style of Structural Operational Semantics (Plotkin, 1981).

3.2.1 Transition Rules

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \quad \text{(Act)} \tag{1}$$

$$\frac{P_j \xrightarrow{\alpha} P'}{P_1 + P_2 \xrightarrow{\alpha} P'} \quad j \in \{1, 2\} \quad \text{(Sum)} \tag{2}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \text{(Par1)} \quad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \quad \text{(Par2)} \tag{3}$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \quad \text{(Com)} \tag{4}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha, \overline{\alpha} \notin L}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \quad \text{(Res)} \tag{5}$$

$$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \quad \text{(Rel)} \tag{6}$$

$$\frac{P \xrightarrow{\alpha} P' \quad K \stackrel{\text{def}}{=} P}{K \xrightarrow{\alpha} P'} \quad \text{(Const)} \tag{7}$$

$$\frac{P[\text{rec } X.P/X] \xrightarrow{\alpha} P'}{\text{rec } X.P \xrightarrow{\alpha} P'} \quad \text{(Rec)} \tag{8}$$

These rules define how CCS processes evolve through actions:

- The (Act) rule states that a process $\alpha.P$ can perform action α and evolve into process P.
- The (Sum) rule allows a sum to behave as any of its components.
- The (Par1) and (Par2) rules allow components of a parallel composition to progress independently.
- The (Com) rule enables synchronization between parallel components.

- The (Res) rule allows actions not in the restriction set to occur.
- The (Rel) rule applies the relabeling function to actions.
- The (Const) rule handles process constants.
- The (Rec) rule unfolds recursive definitions.

3.3 Examples

3.3.1 Simple Communication

Consider the processes:
Coffee = coffee.Coffee Machine = $\overline{\text{coffee}}$.Machine
Their parallel composition can perform a synchronization:
Coffee | Machine $\xrightarrow{\tau}$ Coffee | Machine
This is derived as follows:
(Act) coffee.Coffee $\xrightarrow{\text{coffee}}$ Coffee (Act) $\overline{\text{coffee}}$.Machine $\xrightarrow{\overline{\text{coffee}}}$ Machine (Com) coffee.Coffee | $\overline{\text{coffee}}$.Machine $\xrightarrow{\tau}$ Coffee | Machine

3.3.2 Binary Counter

A binary counter can be represented as follows:
$C_0 = \text{inc}.C_1 + \text{dec}.C_0$ $C_1 = \text{inc}.C_2 + \text{dec}.C_0$ $C_2 = \text{inc}.C_3 + \text{dec}.C_1$ ...
Or more compactly using recursion:
$C_i = \text{inc}.C_{i+1} + \text{dec}.C_{\max(0,i-1)}$

3.3.3 Restriction Example

Considering the coffee machine system with restriction:
(Coffee | Machine) \ {coffee}
The restriction prevents external interaction on the coffee channel, but allows internal communication:
Coffee | Machine $\xrightarrow{\tau}$ Coffee | Machine (Res) (Coffee | Machine)\{coffee} $\xrightarrow{\tau}$ (Coffee | Machine) \ {coffee}
The system can only perform τ actions, making it externally indistinguishable from the process rec $X.\tau.X$.

4 Examples in CAAL and The Need for Program Equivalence

4.1 Examples in CAAL

CAAL (Concurrency Workbench, Aalborg) is a tool for analyzing CCS processes. Here, we examine several classic concurrency problems.

4.1.1 Mutual Exclusion

Mutual exclusion ensures that only one process can access a shared resource at a time. In CCS:

$$\text{User}_i = \text{req}_i.\text{enter}_i.\text{exit}_i.\text{User}_i$$

$$\text{Mutex} = \text{req}_1.\text{enter}_1.\text{exit}_1.\text{Mutex} + \text{req}_2.\text{enter}_2.\text{exit}_2.\text{Mutex}$$

$$\text{System} = (\text{User}_1 \mid \text{User}_2 \mid \text{Mutex}) \setminus \{\text{req}_1, \text{req}_2, \text{exit}_1, \text{exit}_2\}$$

4.1.2 Dining Philosophers

The dining philosophers problem is a classic synchronization problem:

$$\text{Phil}_i = \text{think}_i.\text{get}_{i,left}.\text{get}_{i,right}.\text{eat}_i.\text{put}_{i,left}.\text{put}_{i,right}.\text{Phil}_i$$

$$\text{Fork}_j = \text{get}_{j,right}.\text{put}_{j,right}.\text{Fork}_j + \text{get}_{j+1,left}.\text{put}_{j+1,left}.\text{Fork}_j$$

$$\text{System} = (\text{Phil}_1 \mid \ldots \mid \text{Phil}_n \mid \text{Fork}_1 \mid \ldots \mid \text{Fork}_n) \backslash \{\text{get}_{i,j}, \text{put}_{i,j} \mid 1 \leq i, j \leq n\}$$

4.1.3 Peterson's Algorithm

Peterson's algorithm is a concurrent programming algorithm for mutual exclusion:

$$\text{Process}_i = \text{flag}_i.\text{turn}_{3-i}.\text{wait}_{(flag_{3-i} \wedge turn_{3-i})}.\text{critical}_i.\overline{\text{flag}_i}.\text{Process}_i$$

$$\text{System} = (\text{Process}_1 \mid \text{Process}_2) \setminus \{\text{flag}_1, \text{flag}_2, \text{turn}_1, \text{turn}_2\}$$

4.2 The Need for Program Equivalence

When modeling concurrent systems, we often want to verify that a system implementation meets its specification. For example, we might want to check that a complex implementation of a buffer behaves like the simple buffer we defined earlier.

This requires a formal notion of when two processes are equivalent. However, there are multiple possible notions of equivalence, each capturing different aspects of process behavior:

- **Trace Equivalence**: Two processes are equivalent if they can perform the same sequences of actions.
- **Bisimulation Equivalence**: Two processes are equivalent if they can match each other's behavior step by step.
- **Testing Equivalence**: Two processes are equivalent if they pass the same tests.

- **Failure Equivalence**: Two processes are equivalent if they can perform the same sequences of actions and refuse the same sets of actions after each sequence.

Among these, bisimulation equivalence is particularly important because it captures the branching structure of processes and is a congruence with respect to the operators of CCS.

5 Value-Passing CCS

Value-passing CCS extends basic CCS with the ability to pass data values during communication.

5.1 Syntax Extensions

In value-passing CCS, actions can carry values:

- Input actions: $a(x)$ means "receive a value on channel a and bind it to variable x"
- Output actions: $\overline{a}\langle v \rangle$ means "send value v on channel a"
- Silent action: $\tau.P$ behaves as P without any visible interaction

The syntax is extended as follows:
$P ::= 0 \mid \alpha.P \mid P + P \mid P \mid P \mid P \setminus L \mid P[f] \mid X \mid \text{rec } X.P \mid \sum_{x \in V} P \mid$ if b then P else Q
Where:

- $\sum_{x \in V} P$ represents a choice over a set of values V
- if b then P else Q is a conditional expression
- b is a boolean expression

5.2 Expressions and Variables

Expressions are divided into:

- Arithmetic expressions: $e ::= k \mid x \mid e_1 + e_2 \mid e_1 \times e_2 \mid \ldots$
- Boolean expressions: $b ::= e_1 = e_2 \mid e_1 < e_2 \mid b_1 \wedge b_2 \mid \ldots$

Variables can be bound (have assigned values) or free. A process is closed if all its variables are bound. Only closed processes can be executed.

5.3 Operational Semantics

The operational semantics is extended with rules for value passing:

$$\frac{}{\overline{a}\langle v\rangle.P \xrightarrow{\overline{a}\langle v\rangle} P} \quad \text{(Out)} \tag{9}$$

$$\frac{}{a(x).P \xrightarrow{a(v)} P[v/x]} \quad \text{(In)} \tag{10}$$

$$\frac{P \xrightarrow{\overline{a}\langle v\rangle} P' \quad Q \xrightarrow{a(v)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \quad \text{(Com-Val)} \tag{11}$$

For conditional expressions:

$$\frac{b \text{ evaluates to true}}{\text{if } b \text{ then } P \text{ else } Q \xrightarrow{\tau} P} \quad \text{(If-True)} \tag{12}$$

$$\frac{b \text{ evaluates to false}}{\text{if } b \text{ then } P \text{ else } Q \xrightarrow{\tau} Q} \quad \text{(If-False)} \tag{13}$$

For process constants with parameters:

$$\frac{K(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P \quad e_i \text{ evaluates to } v_i \text{ for } i = 1, \ldots, n}{K(e_1, \ldots, e_n) \xrightarrow{\tau} P[v_1/x_1, \ldots, v_n/x_n]} \quad \text{(Const)} \tag{14}$$

5.4 Encoding in Basic CCS

Value-passing CCS can be encoded in basic CCS by treating each channel-value pair as a separate channel. For example, $a(x).P$ can be encoded as $\sum_{v \in V} a_v.(P[v/x])$, where a_v is a distinct channel for each value v.

Given a set of values $V = \{v_1, v_2, \ldots, v_n\}$, a translation function $[\![\cdot]\!]$: CCS-VP $\to$ CCS can be defined as:

$$
\begin{aligned}
[\![a(x).P]\!] &= \sum_{v \in V} a_v.[\![P[v/x]]\!] \\
[\![\overline{a}\langle e\rangle.P]\!] &= \overline{a}_m.[\![P]\!] \quad \text{if } e \text{ evaluates to } m \\
[\![\tau.P]\!] &= \tau.[\![P]\!] \\
[\![P \mid Q]\!] &= [\![P]\!] \mid [\![Q]\!] \\
[\![P + Q]\!] &= [\![P]\!] + [\![Q]\!] \\
[\![P \setminus L]\!] &= [\![P]\!] \setminus \{a_v \mid a \in L, v \in V\} \\
[\![\text{if } b \text{ then } P \text{ else } Q]\!] &= \begin{cases} [\![P]\!] & \text{if } b \text{ evaluates to true} \\ [\![Q]\!] & \text{otherwise} \end{cases} \\
[\![K(e_1, \ldots, e_n)]\!] &= K_{v_1,\ldots,v_n} \quad \text{if } e_i \text{ evaluates to } v_i \text{ for } i = 1, \ldots, n
\end{aligned}
$$

Where process constants are defined as $K_{v_1,\ldots,v_n} \stackrel{\text{def}}{=} [\![P[v_1/x_1, \ldots, v_n/x_n]]\!]$ if $K(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$.

5.5 Examples

5.5.1 Example: Integer Buffer

A buffer that can store an integer:

IntBuffer = $\text{in}(x).(\text{out}\langle x\rangle.\text{IntBuffer})$

This process receives a value on channel "in", stores it in variable x, then sends the value on channel "out", and returns to its initial state.

5.5.2 Example: Incrementing Buffer

A buffer that increments the value it receives:

$B = \text{in}(x).B'(x)$ $B'(x) = \text{out}\langle x + 1\rangle.B$

5.5.3 Example: Predicate Buffer

A buffer that outputs the predecessor of a number or 0 if the input is 0:

Pred = $\text{in}(x).P(x)$ $P(x) =$ if $x = 0$ then $\text{out}\langle 0\rangle$.Pred else $\text{out}\langle x-1\rangle$.Pred

5.5.4 Example: Counter

A counter that can be incremented and decremented:

$C(x) = \text{inc}.C(x + 1) +$ if $x = 0$ then $\text{dec}.C(0)$ else $\text{dec}.C(x - 1)$

5.5.5 Example: Memory Cell

A one-place memory cell:

Cell = $\text{in}(x).C(x)$ $C(x) = \text{out}\langle x\rangle$.Cell

5.5.6 Example: Two-Place Unordered Buffer

A buffer that can store two values in any order:

$B_2 = \text{in}(x).B_1(x)$ $B_1(x) = \text{out}(x).B_2 + \text{in}(y).B_0(x,y)$ $B_0(x,y) = \text{out}(x).B_1(y) + \text{out}(y).B_1(x)$

5.5.7 Example: FIFO Buffer

A First-In-First-Out buffer with capacity 2:

$F_2 = \text{in}(x).F_1(x)$ $F_1(x) = \text{out}(x).F_2 + \text{in}(y).F_0(x,y)$ $F_0(x,y) = \text{out}(x).F_1(y)$

6 (Strong) Bisimilarity

Bisimilarity is a notion of equivalence between processes that captures the idea that two processes behave the same way.

6.1 Definition

A binary relation R on processes is a bisimulation if whenever $(P, Q) \in R$:

1. If $P \xrightarrow{\alpha} P'$, then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in R$
2. If $Q \xrightarrow{\alpha} Q'$, then there exists P' such that $P \xrightarrow{\alpha} P'$ and $(P', Q') \in R$

Two processes P and Q are bisimilar, written $P \sim Q$, if there exists a bisimulation R such that $(P, Q) \in R$.

6.2 Examples

1. $a.b.0 + a.c.0 \not\sim a.(b.0 + c.0)$

   The first process can make a choice after performing the a action, while the second process makes the choice before performing the a action.

2. $a.(b.0 + c.0) \sim a.b.0 + a.c.0 + a.(b.0 + c.0)$

   The third term is redundant because its behavior is already covered by the first two terms.

7 (Strong) Bisimilarity and its Game-Theoretical Characterization

7.1 Bisimulation Game

Bisimilarity can be characterized as a two-player game:

- Players: Attacker and Defender

- Initial state: Two processes P and Q
- Rounds:
  1. Attacker chooses one process (either P or Q) and one of its possible transitions $\xrightarrow{\alpha}$
  2. Defender must match this transition with the other process
  3. The game continues with the resulting processes
- Outcome:
  - Defender wins if they can always match the Attacker's moves
  - Attacker wins if at some point Defender cannot match their move

Two processes are bisimilar if and only if Defender has a winning strategy in the bisimulation game.

7.2 Properties of Bisimilarity

7.2.1 Largest Bisimulation

The relation $\sim$ is the largest bisimulation, i.e., it contains all other bisimulations.

If R is a bisimulation, then $R \subseteq \sim$.

7.2.2 Equivalence

Bisimilarity is an equivalence relation:

The relation $\sim$ is reflexive, symmetric, and transitive.

7.3 Finite CCS

Finite CCS is the fragment of CCS without recursion. Every finite CCS process has a finite labeled transition system.

For any two finite CCS processes P and Q, it is decidable whether $P \sim Q$.

The algorithm to decide bisimilarity of finite processes is based on the partition refinement technique.

8 Properties of Bisimilarity: Exercises and Compositionality

8.1 Exercises

1. Show that $a.0 \mid b.0 \sim a.b.0 + b.a.0$

2. Show that $a.b.0 + a.c.0 \not\sim a.(b.0 + c.0)$

3. Show that $(a.0 \mid b.0) \setminus \{a\} \sim 0 \mid b.0$

8.2 Compositionality

A relation R is a congruence if it is preserved by all operators of the language. Bisimilarity is a congruence for CCS:

If $P \sim Q$, then:

1. $\alpha.P \sim \alpha.Q$

2. $P + R \sim Q + R$

3. $P \mid R \sim Q \mid R$

4. $P \setminus L \sim Q \setminus L$

5. $P[f] \sim Q[f]$

Compositionality is a crucial property for verification: it allows us to verify components in isolation and then compose them.

9 Weak Bisimilarity: Definition

Strong bisimilarity treats the internal action τ like any other action. However, since τ is not observable, we might want a coarser equivalence that abstracts away from internal actions.

9.1 Weak Transitions

Define the weak transition $\Rightarrow$ as:

$$P \Rightarrow P' \text{ if } P \xrightarrow{\tau^*} P' \tag{15}$$

$$P \overset{\alpha}{\Rightarrow} P' \text{ if } P \Rightarrow \xrightarrow{\alpha} \Rightarrow P' \text{ for } \alpha \neq \tau \tag{16}$$

$$P \overset{\tau}{\Rightarrow} P' \text{ if } P \Rightarrow P' \text{ (possibly with no } \tau \text{ transitions)} \tag{17}$$

where $\xrightarrow{\tau^*}$ means zero or more τ transitions.

9.2 Definition

A binary relation R on processes is a weak bisimulation if whenever $(P, Q) \in R$:

1. If $P \xrightarrow{\alpha} P'$, then there exists Q' such that $Q \overset{\alpha}{\Rightarrow} Q'$ and $(P', Q') \in R$

2. If $Q \xrightarrow{\alpha} Q'$, then there exists P' such that $P \stackrel{\alpha}{\Longrightarrow} P'$ and $(P', Q') \in R$

Two processes P and Q are weakly bisimilar, written $P \approx Q$, if there exists a weak bisimulation R such that $(P, Q) \in R$.

9.3 Examples

9.3.1 Basic Examples

1. $\tau.a.0 \approx a.0$

   The internal action τ is ignored in weak bisimilarity.

2. $a.(\tau.b.0 + \tau.c.0) \approx a.b.0 + a.c.0$

   The internal choice after the a action is equivalent to an external choice before it.

9.3.2 FIFO Buffer Example

Consider a FIFO buffer with capacity 2 modeled in two different ways:

$$\begin{aligned}
\text{Cell} &= \text{in}(a).\text{C}(a) \\
\text{C}(a) &= \text{out}(a).\text{Cell} \\
\text{F}_2 &= \text{in}(x).\text{F}_1(x) \\
\text{F}_1(x) &= \text{out}(x).\text{F}_2 + \text{in}(y).\text{F}_0(x, y) \\
\text{F}_0(x, y) &= \text{out}(x).\text{F}_1(y)
\end{aligned}$$

We can prove that $(\text{Cell} \mid \text{Cell})\backslash\{\text{in}, \text{out}\} \approx \text{F}_2$. To do this, we would need to establish a weak bisimulation R such that $((\text{Cell} \mid \text{Cell})\backslash\{\text{in}, \text{out}\}, \text{F}_2) \in R$.

9.3.3 Non-lossy Channel over a Lossy Channel

Consider a system that implements a reliable channel over an unreliable medium:

$$\begin{aligned}
\text{SEND} &= \text{in}(x).\text{SENDING}(x) \\
\text{SENDING}(x) &= \overline{\text{send}}(x).\text{WAIT}(x) \\
\text{WAIT}(x) &= \text{error}.\text{SENDING}(x) + \text{ack}.\text{SEND} \\
\text{MED} &= \text{send}(x).\text{MED}'(x) \\
\text{MED}'(x) &= \tau.\text{error}.\text{MED} + \overline{\text{trans}}(x).\text{MED} \\
\text{REC} &= \text{trans}(x).\text{DEL}(x) \\
\text{DEL}(x) &= \overline{\text{out}}(x).\overline{\text{ack}}.\text{REC}
\end{aligned}$$

The complete system is:

$$\text{SYS} = (\text{SEND} \mid \text{MED} \mid \text{REC}) \setminus \{\text{send}, \text{trans}, \text{error}, \text{ack}\}$$

It can be shown that this system is weakly bisimilar to the simple specification:

$$\text{SPEC} = \text{in}(x).\overline{\text{out}}(x).\text{SPEC}$$

This demonstrates that the complex implementation, despite having potential for errors and retransmissions, behaves externally just like a simple reliable channel.

10 Weak Bisimilarity: Properties

10.1 Relation to Strong Bisimilarity

Strong bisimilarity implies weak bisimilarity, but not vice versa:

If $P \sim Q$, then $P \approx Q$.

10.2 Equivalence

Like strong bisimilarity, weak bisimilarity is an equivalence relation:

The relation $\approx$ is reflexive, symmetric, and transitive.

10.3 Compositionality

Unlike strong bisimilarity, weak bisimilarity is not a congruence for all CCS operators. In particular, it is not preserved by the + operator:

$\tau.a.0 \approx a.0$, but $\tau.a.0 + b.0 \not\approx a.0 + b.0$

However, weak bisimilarity is a congruence for the other CCS operators:

If $P \approx Q$, then:

1. $\alpha.P \approx \alpha.Q$
2. $P \mid R \approx Q \mid R$
3. $P \setminus L \approx Q \setminus L$
4. $P[f] \approx Q[f]$

10.4 Observational Congruence

To address the lack of congruence with respect to the + operator, a stronger equivalence called observational congruence is defined:

$P \cong Q$ if:

1. If $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \xrightarrow{\tau}\Rightarrow Q'$ and $P' \approx Q'$

2. If $Q \xrightarrow{\tau} Q'$, then there exists P' such that $P \xrightarrow{\tau}\Rightarrow P'$ and $P' \approx Q'$
3. If $P \xrightarrow{\alpha} P'$ with $\alpha \neq \tau$, then there exists Q' such that $Q \xRightarrow{\alpha} Q'$ and $P' \approx Q'$
4. If $Q \xrightarrow{\alpha} Q'$ with $\alpha \neq \tau$, then there exists P' such that $P \xRightarrow{\alpha} P'$ and $P' \approx Q'$

Observational congruence is a congruence for all CCS operators, including the + operator.

11 Fixed Point Theory

Fixed point theory provides the mathematical foundation for understanding recursive processes in CCS.

11.1 Complete Partial Orders

A partially ordered set (poset) $(D, \sqsubseteq)$ is a complete partial order (CPO) if:

1. It has a least element $\bot$ (bottom)
2. Every directed subset has a least upper bound (lub)

A subset $S \subseteq D$ is directed if for any $x, y \in S$, there exists $z \in S$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$.

11.2 Continuous Functions

A function $f : D \to E$ between CPOs is continuous if:

1. It is monotonic: if $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$
2. It preserves lubs of directed sets: $f(\bigsqcup S) = \bigsqcup f(S)$ for any directed set S

11.3 Fixed Points

A fixed point of a function $f : D \to D$ is an element $x \in D$ such that $f(x) = x$.

[Kleene's Fixed Point Theorem] Every continuous function $f : D \to D$ on a CPO D has a least fixed point, given by:

$$\text{fix}(f) = \bigsqcup_{n \geq 0} f^n(\bot)$$

11.4 Powerset Lattices

For a set S, the powerset $\mathcal{P}(S)$ with the subset inclusion order $\subseteq$ forms a complete lattice:

1. The least element is the empty set $\emptyset$
2. The greatest element is S itself
3. The lub of a family of sets is their union
4. The greatest lower bound (glb) of a family of sets is their intersection

12 Bisimilarity as a Fixed Point

12.1 The Function $\mathcal{F}$

Let $\mathcal{P}(P \times P)$ be the set of all binary relations on processes, ordered by set inclusion. Define the function $\mathcal{F} : \mathcal{P}(P \times P) \to \mathcal{P}(P \times P)$ as:

$\mathcal{F}(R) = \{(P, Q) \mid$ if $P \xrightarrow{\alpha} P'$ then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in R$, and if $Q \xrightarrow{\alpha} Q'$ then there exists P' such that $P \xrightarrow{\alpha} P'$ and $(P', Q') \in R\}$

12.2 Bisimilarity as Greatest Fixed Point

The bisimilarity relation $\sim$ is the greatest fixed point of $\mathcal{F}$.

This characterization leads to a proof technique called coinduction:

[Coinduction Principle] To show $P \sim Q$, it suffices to find a bisimulation R such that $(P, Q) \in R$.

12.3 Algorithm for Finite Systems

For finite labeled transition systems, bisimilarity can be computed using the partition refinement algorithm:

1. Start with a single partition containing all states
2. Iteratively refine the partition based on the transitions
3. The algorithm terminates when no further refinement is possible
4. States in the same final partition are bisimilar

This algorithm has time complexity $O(mn \log n)$, where m is the number of transitions and n is the number of states.

13 Hennessy-Milner Logic

13.1 Syntax

Hennessy-Milner Logic (HML) is a modal logic for specifying properties of processes:

$\phi ::= \text{true} \mid \neg\phi \mid \phi \wedge \phi \mid \langle\alpha\rangle\phi$

Where:

- true is always satisfied
- $\neg\phi$ is satisfied if ϕ is not satisfied
- $\phi \wedge \psi$ is satisfied if both ϕ and ψ are satisfied
- $\langle\alpha\rangle\phi$ is satisfied if there is an α-transition to a state satisfying ϕ

Additional operators can be defined as abbreviations:

- $\text{false} \equiv \neg\text{true}$
- $\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$
- $[\alpha]\phi \equiv \neg\langle\alpha\rangle\neg\phi$ (meaning all α-transitions lead to states satisfying ϕ)

13.2 Semantics

The satisfaction relation $\models$ is defined inductively:

$$P \models \text{true for all } P \tag{18}$$
$$P \models \neg\phi \text{ iff } P \not\models \phi \tag{19}$$
$$P \models \phi \wedge \psi \text{ iff } P \models \phi \text{ and } P \models \psi \tag{20}$$
$$P \models \langle\alpha\rangle\phi \text{ iff there exists } P' \text{ such that } P \xrightarrow{\alpha} P' \text{ and } P' \models \phi \tag{21}$$

We can define the semantic function $[\![\cdot]\!] : \text{HML} \to 2^{\text{Proc}}$ as:

$$[\![\text{true}]\!] = \text{Proc} \tag{22}$$
$$[\![\neg\phi]\!] = \text{Proc} \setminus [\![\phi]\!] \tag{23}$$
$$[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!] \tag{24}$$
$$[\![\langle\alpha\rangle\phi]\!] = \{P \mid \exists P'.P \xrightarrow{\alpha} P' \text{ and } P' \in [\![\phi]\!]\} \tag{25}$$

Then, $P \models \phi$ if and only if $P \in [\![\phi]\!]$.

13.3 Examples

13.3.1 Basic Properties

- "Can perform a coffee action": $\langle$coffee$\rangle$true
- "Cannot perform a coffee action": $\neg\langle$coffee$\rangle$true or equivalently [coffee]false
- "After a coffee action, can perform a tea action": $\langle$coffee$\rangle\langle$tea$\rangle$true
- "After any coffee action, must be able to perform a tea action": [coffee]$\langle$tea$\rangle$true

13.3.2 Distinguishing Non-Bisimilar Processes

Consider the processes:

$$\begin{aligned} S &= a.S_1 + a.S_2 \\ S_1 &= b.0 \\ S_2 &= c.0 \\ T &= a.T_1 \\ T_1 &= b.0 + c.0 \end{aligned}$$

These processes are not bisimilar. The distinguishing formula is: $\phi = \langle a\rangle([\text{b}]\text{false} \wedge [\text{c}]\text{true})$

We have $S \models \phi$ (via the path to S_2) but $T \not\models \phi$.

13.4 Characterizing Bisimilarity

HML characterizes bisimilarity in the following sense:

[Hennessy-Milner Theorem] For image-finite processes (processes with a finite number of α-derivatives for each action α): $P \sim Q$ iff for all HML formulas ϕ : $P \models \phi$ iff $Q \models \phi$

This theorem establishes a deep connection between operational and logical views of processes.

13.4.1 Proof Sketch

1. First direction ($\Rightarrow$): If $P \sim Q$, then for all HML formulas ϕ, $P \models \phi$ iff $Q \models \phi$. This can be proven by induction on the structure of ϕ.

2. Second direction ($\Leftarrow$): Define the relation $R = \{(P, Q) \mid \text{for all } \phi, P \models \phi \text{ iff } Q \models \phi\}$. We show that R is a bisimulation.

   Suppose $(P, Q) \in R$ and $P \xrightarrow{\alpha} P'$. We need to show that there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in R$.

   By contradiction, suppose no such Q' exists. Due to image-finiteness, there are finitely many α-derivatives of Q, say $Q'_1, Q'_2, \ldots, Q'_n$. For

each Q'_i, there must exist a formula ϕ_i such that $P' \models \phi_i$ but $Q'_i \not\models \phi_i$. Define $\phi = \phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n$.

Now $P \models \langle\alpha\rangle\phi$ but $Q \not\models \langle\alpha\rangle\phi$, contradicting $(P, Q) \in R$.

13.4.2 Counterexample for Non-Image-Finite Processes

The Hennessy-Milner theorem does not hold for non-image-finite processes. As a counterexample, consider:

$$A_\omega = \sum_{i \in \mathbb{N}} a^i.0$$
$$A'_\omega = A_\omega + a^\omega$$

Where a^i denotes i consecutive a actions and a^ω denotes an infinite sequence of a actions. These processes satisfy the same HML formulas but are not bisimilar.

14 Logic with Recursion

14.1 Introduction

HML cannot express temporal properties like "eventually" or "always". To address this limitation, we extend HML with recursion.

14.2 Example Properties

- "Eventually a": The process will eventually perform an a action
- "Always not a": The process will never perform an a action
- "Liveness": The process can always perform some action
- "Fairness": If an action is enabled infinitely often, it will eventually be performed

14.3 Syntax and Semantics

The syntax of the modal μ-calculus extends HML with fixed point operators:

$$\phi ::= \text{true} \mid \neg\phi \mid \phi \wedge \phi \mid \langle\alpha\rangle\phi \mid X \mid \mu X.\phi$$

Where:

- X is a variable
- $\mu X.\phi$ is the least fixed point of $\lambda X.\phi$

The greatest fixed point operator $\nu X.\phi$ can be defined as $\neg\mu X.\neg\phi[\neg X/X]$.

The semantics is defined with respect to an environment ρ that maps variables to sets of processes:

$$[\![\text{true}]\!]_\rho = P \tag{26}$$
$$[\![\neg\phi]\!]_\rho = P \setminus [\![\phi]\!]_\rho \tag{27}$$
$$[\![\phi \wedge \psi]\!]_\rho = [\![\phi]\!]_\rho \cap [\![\psi]\!]_\rho \tag{28}$$
$$[\![\langle\alpha\rangle\phi]\!]_\rho = \{P \mid \exists P'.P \xrightarrow{\alpha} P' \text{ and } P' \in [\![\phi]\!]_\rho\} \tag{29}$$
$$[\![X]\!]_\rho = \rho(X) \tag{30}$$
$$[\![\mu X.\phi]\!]_\rho = \text{lfp}(F) \text{ where } F(S) = [\![\phi]\!]_{\rho[X\mapsto S]} \tag{31}$$

14.4 Examples

- "Eventually a": $\mu X.(\langle a\rangle\text{true} \vee \langle\tau\rangle X)$
- "Always not a": $\nu X.([\alpha]\text{false} \wedge [\tau]X)$
- "Liveness": $\nu X.(\langle\tau\rangle\text{true} \vee \exists\alpha \neq \tau.\langle\alpha\rangle\text{true}) \wedge [\tau]X$
- "Deadlock freedom": $\nu X.(\langle-\rangle\text{true} \wedge [-]X)$

15 Verifying Properties with the Edinburgh CWB

15.1 Edinburgh Concurrency Workbench

The Edinburgh Concurrency Workbench (CWB) is a tool for analyzing concurrent systems. It supports:

- Process description in CCS
- Equivalence checking (bisimulation, trace, testing)
- Model checking of modal μ-calculus properties
- Simulation and state exploration

15.2 Verifying Properties

15.2.1 Mutual Exclusion

Mutual exclusion requires that two processes never access the critical section simultaneously:

$$\text{ME} = \nu X.([-]X \wedge \neg\langle\text{enter}_1\rangle\langle\text{enter}_2\rangle\text{true})$$

15.2.2 Fairness

Strong fairness requires that if an action is enabled infinitely often, it will eventually be performed:

$$\text{SF}(a) = \nu Y.\mu X.(([-]\text{false} \vee \langle a \rangle \text{true}) \vee ([-]X \wedge \nu Z.(\langle - \rangle \text{true} \wedge [a]\text{false} \wedge [-]Z) \Rightarrow [-]Y))$$

15.2.3 Liveness

Liveness requires that the system can always make progress:

$$\text{Live} = \nu X.(\langle - \rangle \text{true} \wedge [-]X)$$

15.2.4 Deadlock Freedom

Deadlock freedom requires that the system never reaches a state where no actions are possible:

$$\text{DF} = \nu X.(\langle - \rangle \text{true} \wedge [-]X)$$

16 Pi-Calculus

16.1 From CCS to π-calculus

The π-calculus, developed by Robin Milner, Joachim Parrow, and David Walker, extends CCS with the ability to communicate channel names. This allows for dynamic reconfiguration of the communication topology.

16.2 Syntax

$$P ::= 0 \mid \pi.P \mid P + P \mid P \mid P \mid \nu x.P \mid !P$$

Where:

- π is a prefix, which can be:
  - $x(y)$: Receive a name on channel x and bind it to y
  - $\overline{x}\langle y \rangle$: Send the name y on channel x
  - τ: Silent action
- $\nu x.P$: Create a new name x with scope P
- $!P$: Replication (infinitely many parallel copies of P)

16.3 Operational Semantics

$$\frac{}{\pi.P \xrightarrow{\pi} P} \quad \text{(Prefix)} \tag{32}$$

$$\frac{P \xrightarrow{\pi} P'}{P + Q \xrightarrow{\pi} P'} \quad \text{(Sum)} \tag{33}$$

$$\frac{P \xrightarrow{\pi} P'}{P \mid Q \xrightarrow{\pi} P' \mid Q} \quad \text{(Par)} \tag{34}$$

$$\frac{P \xrightarrow{x(z)} P' \quad Q \xrightarrow{\overline{x}\langle y \rangle} Q'}{P \mid Q \xrightarrow{\tau} P'[y/z] \mid Q'} \quad \text{(Com)} \tag{35}$$

$$\frac{P \xrightarrow{\pi} P' \quad x \notin \text{fn}(\pi)}{\nu x.P \xrightarrow{\pi} \nu x.P'} \quad \text{(Res)} \tag{36}$$

$$\frac{P \xrightarrow{\overline{y}\langle z \rangle} P' \quad x \neq y \quad x = z \text{ or } x \in \text{fn}(P')}{\nu x.P \xrightarrow{\overline{y}\langle \nu x \rangle} P'} \quad \text{(Open)} \tag{37}$$

$$\frac{P \mid !P \xrightarrow{\pi} P'}{!P \xrightarrow{\pi} P'} \quad \text{(Rep)} \tag{38}$$

16.4 Example: Mobile Telephones

A model of a mobile telephone network, where a telephone can move between different base stations:

$$\text{Telephone}(id, base) = \overline{base}\langle id \rangle.\text{base}(new\_base).\text{Telephone}(id, new\_base)$$

$$\text{BaseStation}(i) = id(tel\_id).\overline{tel\_id}\langle base_j \rangle.\text{BaseStation}(i)$$

$$\text{Network} = \nu base_1, \ldots, base_n.(\text{Telephone}(id_1, base_1) \mid \ldots \mid \text{BaseStation}(1) \mid \ldots)$$

16.5 From Modelling to Programming Languages

Process calculi like CCS and π-calculus have influenced the design of programming languages with built-in concurrency features, like Google Go, Erlang, and Clojure, which we will study next.

17 Google Go: Basics

17.1 Introduction to Go

Go is a statically typed, compiled language with built-in concurrency features, designed by Google. Key features include:

- Simple syntax inspired by C
- Garbage collection
- Concurrency primitives (goroutines, channels)
- Strong typing with type inference
- Methods and interfaces for object-oriented programming

17.2 Basic Syntax

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello, world!")
}
```

17.3 Types and Data Structures

```
// Basic types
var i int = 42
var f float64 = 3.14
var s string = "hello"
var b bool = true

// Arrays and slices
var arr [5]int = [5]int{1, 2, 3, 4, 5}
var slc []int = []int{1, 2, 3}
slc = append(slc, 4, 5)

// Maps
var m map[string]int = map[string]int{
    "one": 1,
    "two": 2,
}
```

17.4 Control Structures

```
// If statement
if x > 0 {
    fmt.Println("Positive")
} else if x < 0 {
    fmt.Println("Negative")
} else {
    fmt.Println("Zero")
}

// For loop
for i := 0; i < 10; i++ {
    fmt.Println(i)
}

// Range loop
for i, v := range slc {
    fmt.Printf("Index: %d, Value: %d\n", i, v)
}

// Switch statement
switch day {
case "Monday":
    fmt.Println("Start of work week")
case "Friday":
    fmt.Println("End of work week")
default:
    fmt.Println("Regular day")
}
```

17.5 Functions and Methods

```
// Function
func add(a, b int) int {
    return a + b
}

// Multiple return values
func divMod(a, b int) (int, int) {
    return a / b, a % b
}

// Named return values
func divModNamed(a, b int) (quotient, remainder int) {
    quotient = a / b
    remainder = a % b
    return
```

```
}

// Variadic function
func sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
    return total
}

// Method
type Rectangle struct {
    width, height float64
}

func (r Rectangle) Area() float64 {
    return r.width * r.height
}
```

17.6 Interfaces

```
// Interface
type Shape interface {
    Area() float64
}

type Circle struct {
    radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * c.radius * c.radius
}

// Both Rectangle and Circle implement the Shape
   interface
func PrintArea(s Shape) {
    fmt.Printf("Area: %f\n", s.Area())
}

func main() {
    r := Rectangle{width: 3, height: 4}
    c := Circle{radius: 5}

    PrintArea(r)
    PrintArea(c)
}
```

18 Google Go: Concurrency. Part 1

18.1 Goroutines

Goroutines are lightweight threads managed by the Go runtime:

```
func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

18.2 Channels

Channels are typed conduits for sending and receiving values:

```
func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // Send sum to channel c
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)

    x, y := <-c, <-c // Receive from channel c

    fmt.Println(x, y, x+y)
}
```

18.3 Synchronous vs. Asynchronous Communication

By default, channels in Go are synchronous (unbuffered), meaning sending and receiving operations block until the other side is ready:

```
// Synchronous channel
ch := make(chan int)
```

```
// Asynchronous (buffered) channel with capacity 10
bufferedCh := make(chan int, 10)
```

18.4 Select

The select statement lets a goroutine wait on multiple communication operations:

```
func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)

    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()

    fibonacci(c, quit)
}
```

18.5 Timeouts

The select statement can be used with timeouts:

```
func main() {
    c := make(chan int)

    go func() {
        time.Sleep(2 * time.Second)
        c <- 1
    }()
```

```
    select {
    case res := <-c:
        fmt.Println(res)
    case <-time.After(1 * time.Second):
        fmt.Println("timeout 1")
    }
}
```

19 Google Go: Concurrency. Part 2

19.1 Using Select and Timeouts

Select with timeouts can be used for implementing non-blocking operations and handling slow operations:

```
func main() {
    c1 := make(chan string, 1)
    go func() {
        time.Sleep(2 * time.Second)
        c1 <- "result 1"
    }()

    select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(1 * time.Second):
        fmt.Println("timeout 1")
    }
}
```

19.2 Concurrency and Shared Memory

Go follows the philosophy "Do not communicate by sharing memory; instead, share memory by communicating." However, it also provides traditional synchronization primitives:

```
var mu sync.Mutex
var count int

func increment() {
    mu.Lock()
    count++
    mu.Unlock()
}

func main() {
    var wg sync.WaitGroup
```

```
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            increment()
        }()
    }

    wg.Wait()
    fmt.Println("Count:", count)
}
```

19.3 Channels of Channels

Channels are first-class values and can be passed on channels themselves, enabling sophisticated concurrency patterns:

19.3.1 Server

```
type Request struct {
    args        []int
    resultChan  chan int
}

func sum(args []int) int {
    sum := 0
    for _, v := range args {
        sum += v
    }
    return sum
}

func server(reqChan chan Request) {
    for req := range reqChan {
        req.resultChan <- sum(req.args)
    }
}

func main() {
    reqChan := make(chan Request)
    go server(reqChan)

    request := Request{
        args:       []int{3, 4, 5},
        resultChan: make(chan int),
    }

    reqChan <- request
```

```
    fmt.Println(<-request.resultChan) // 12
}
```

19.3.2 Task Splitting

```
func worker(id int, jobs <-chan int, results chan<- int)
    {
    for j := range jobs {
        fmt.Printf("Worker %d started job %d\n", id, j)
        time.Sleep(time.Second)
        fmt.Printf("Worker %d finished job %d\n", id, j)
        results <- j * 2
    }
}

func main() {
    const numJobs = 5
    jobs := make(chan int, numJobs)
    results := make(chan int, numJobs)

    // Start workers
    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }

    // Send jobs
    for j := 1; j <= numJobs; j++ {
        jobs <- j
    }
    close(jobs)

    // Collect results
    for a := 1; a <= numJobs; a++ {
        <-results
    }
}
```

19.3.3 Replication

```
func main() {
    c := make(chan int)

    // Replicate the message to multiple receivers
    for i := 0; i < 5; i++ {
        go func(i int) {
            fmt.Printf("Receiver %d got: %d\n", i, <-c)
        }(i)
```

```
    }

    c <- 42 // Only one receiver will get this

    // Fan-out with multiple channels
    channels := make([]chan int, 5)
    for i := range channels {
        channels[i] = make(chan int)
        go func(i int, ch chan int) {
            fmt.Printf("Dedicated receiver %d got: %d\n",
                i, <-ch)
        }(i, channels[i])
    }

    for i, ch := range channels {
        ch <- i * 10
    }

    time.Sleep(time.Second)
}
```

19.3.4 Pipeline

```
func gen(nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        for _, n := range nums {
            out <- n
        }
        close(out)
    }()
    return out
}

func square(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for n := range in {
            out <- n * n
        }
        close(out)
    }()
    return out
}

func main() {
    c := gen(2, 3, 4)
    out := square(c)
```

```
    for n := range out {
        fmt.Println(n) // 4, 9, 16
    }
}
```

20 Google Go: Concurrency. Part 3

20.1 Low Level Concurrency and Memory Model

Go's memory model defines when one goroutine can observe the effects of another:

```
var a, b int

func f() {
    a = 1
    b = 2
}

func g() {
    print(b)
    print(a)
}

func main() {
    go f()
    g()
}
```

In this example, g might print "0 0", "2 0", or "2 1", but not "0 1" because there's a happens-before relationship between the assignment to a and b.

20.2 The Scheduler

Go's runtime includes a sophisticated scheduler that manages goroutines:

- Work-stealing scheduler
- Cooperative scheduling with preemption
- Integration with the garbage collector
- GOMAXPROCS determines the number of OS threads

```
func main() {
    fmt.Printf("GOMAXPROCS: %d\n", runtime.GOMAXPROCS(0))

    // Change GOMAXPROCS
    runtime.GOMAXPROCS(4)
    fmt.Printf("GOMAXPROCS: %d\n", runtime.GOMAXPROCS(0))
}
```

20.3 Concluding Remarks

Go's approach to concurrency is based on CSP (Communicating Sequential Processes), with channels as the primary means of synchronization. This design choice leads to more maintainable concurrent code compared to traditional shared-memory concurrency.

20.4 Erlang: Some Basic Ideas

Erlang is a functional programming language with built-in support for concurrency, distribution, and fault tolerance. Key features include:

- Actor model of concurrency
- Immutable data
- Pattern matching
- Hot code swapping
- "Let it crash" philosophy

21 Erlang: Basic Features

21.1 Functional Programming

Erlang is a functional programming language with:

- Immutable variables
- Pattern matching
- Higher-order functions
- Recursion instead of loops

```erlang
% Basic syntax
X = 42.  % Variable binding
Y = X + 1.  % Y is 43
% X = 50.  % Would throw an error, variables are
    immutable

% Pattern matching
case {1, 2, 3} of
    {1, X, 3} -> X;  % Matches, returns 2
    {2, _, _} -> error;
    _ -> default
end.

% Functions
factorial(0) -> 1;
factorial(N) when N > 0 -> N * factorial(N-1).

% Higher-order functions
Double = fun(X) -> X * 2 end.
Lists:map(Double, [1, 2, 3]).  % Returns [2, 4, 6]

% List comprehensions
[X * 2 || X <- [1, 2, 3]].  % Returns [2, 4, 6]
```

21.2 Concurrency: Spawn, Send, Receive

Erlang's concurrency is based on the actor model:

```erlang
% Spawn a process
Pid = spawn(fun() ->
    io:format("Hello from process ~p~n", [self()])
end).

% Send a message
Pid ! {hello, "world"}.

% Receive a message
receive
    {hello, Msg} ->
        io:format("Received: ~p~n", [Msg]);
    Other ->
        io:format("Unexpected: ~p~n", [Other])
after 1000 ->
    io:format("Timeout~n")
end.
```

21.3 Example: Echo Server

```
% Echo server
echo() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            echo();
        stop ->
            ok
    end.

% Usage
Server = spawn(fun echo/0).
Server ! {self(), "Hello"}.
receive
    {Server, Msg} ->
        io:format("Echo: ~p~n", [Msg])
end.
Server ! stop.
```

22 Erlang: Robustness, Distribution, Hotswapping

22.1 Robustness

Erlang's "let it crash" philosophy is supported by process linking and monitoring:

```
% Link processes
spawn_link(fun() ->
    % This will crash the parent process too
    1 = 2  % Deliberate error
end).

% Process monitoring
Pid = spawn(fun() -> timer:sleep(1000) end).
Ref = monitor(process, Pid).
receive
    {'DOWN', Ref, process, Pid, Reason} ->
        io:format("Process ~p died: ~p~n", [Pid, Reason])
end.

% Supervision tree
start_link() ->
    supervisor:start_link({local, my_sup}, ?MODULE, []).

init([]) ->
    ChildSpecs = [
        {my_server, {my_server, start_link, []},
         permanent, 5000, worker, [my_server]}
```

```
    ],
    {ok, {{one_for_one, 5, 10}, ChildSpecs}}.
```

22.2 Distribution

Erlang processes can communicate across different machines in a transparent way:

```
% Start a node
erl -name node1@192.168.1.100

% Connect to another node
net_adm:ping('node2@192.168.1.101').

% Spawn a process on a remote node
Pid = spawn('node2@192.168.1.101', fun() ->
    io:format("Hello from ~p~n", [node()])
end).

% Send a message to a remote process
Pid ! {hello, "remote"}.

% Global registry
global:register_name(my_service, self()).
Pid = global:whereis_name(my_service).
```

22.3 Hot Code Swapping

Erlang supports upgrading code without stopping the system:

```
% Version 1
-module(server).
-export([start/0, loop/0, upgrade/1]).

start() ->
    spawn(fun() -> loop() end).

loop() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            loop();
        {upgrade, Fun} ->
            Fun()
    end.

upgrade(Pid) ->
    Pid ! {upgrade, fun() -> ?MODULE:loop_v2() end}.
```

```
% Version 2
loop_v2() ->
    receive
        {From, Msg} ->
            From ! {self(), {v2, Msg}},
            loop_v2();
        {upgrade, Fun} ->
            Fun()
    end.
```

The key to hot code swapping is the fully qualified call '?MODULE:loop$_v$2()', *whichensuresthatthe*

23 Clojure: Basics, Functional Parallelism, Laziness

23.1 Basics of Clojure

Clojure is a Lisp dialect that runs on the JVM and emphasizes functional programming with immutable data structures:

```
;; Basic syntax
(def x 42)
(def y (+ x 1))  ; y is 43

;; Functions
(defn square [x]
  (* x x))

(square 5)  ; Returns 25

;; Anonymous functions
(def cube #(* % % %))
(cube 3)  ; Returns 27

;; Higher-order functions
(map #(* 2 %) [1 2 3])  ; Returns (2 4 6)

;; Immutable data structures
(def v [1 2 3])
(def v2 (conj v 4))  ; v2 is [1 2 3 4], v is still [1 2
   3]

;; Destructuring
(let [[a b & rest] [1 2 3 4 5]]
  [a b rest])  ; Returns [1 2 (3 4 5)]
```

23.2 Functional Parallelism

Clojure provides several constructs for parallel programming:

```
;; pmap: parallel map
(def numbers (range 1000000))
(time (doall (map #(Thread/sleep 10) (range 10))))
(time (doall (pmap #(Thread/sleep 10) (range 10))))

;; reducers: parallel reduce
(require '[clojure.core.reducers :as r])
(time (reduce + (map inc (range 1000000))))
(time (r/fold + (r/map inc (range 1000000))))
```

23.3 Laziness

Clojure sequences are lazy by default, meaning they are computed only as needed:

```
;; Infinite sequence
(def natural-numbers (iterate inc 1))
(take 10 natural-numbers)  ; Returns (1 2 3 4 5 6 7 8 9
    10)

;; Lazy sequence with side effects
(def numbers-with-side-effect
  (map #(do (println %) %) (range 10)))
;; Nothing is printed until we consume the sequence
(doall (take 5 numbers-with-side-effect))
;; Prints 0 1 2 3 4 and returns (0 1 2 3 4)

;; Lazy file processing
(with-open [rdr (clojure.java.io/reader "large-file.txt")
    ]
  (doseq [line (line-seq rdr)]
    (when (re-find #"important" line)
      (println line))))
```

24 Clojure: Futures and Promises, Shared Mutable State

24.1 Futures and Promises

Futures and promises provide a way to work with values that may not yet be available:

```
;; Future: executes a task in a separate thread
(def f (future
```

```
          (Thread/sleep 1000)
          (+ 1 2)))
;; Blocks until the result is available
(deref f)  ; or @f, returns 3

;; Promise: a placeholder for a value to be delivered
   later
(def p (promise))
(def t (future
          (Thread/sleep 1000)
          (deliver p 42)))
;; Blocks until a value is delivered
(deref p 2000 :timeout)  ; Returns 42, or :timeout if it
   takes > 2 seconds
```

24.2 Shared Mutable State

Clojure provides several reference types for managing shared mutable state:

```
;; Atom: synchronous, uncoordinated reference
(def counter (atom 0))
(swap! counter inc)  ; Atomically increments the counter
(reset! counter 0)   ; Sets the counter to 0

;; Ref: synchronous, coordinated reference
(def account1 (ref 1000))
(def account2 (ref 500))
(dosync
  (alter account1 - 100)
  (alter account2 + 100))

;; Agent: asynchronous reference
(def log (agent []))
(send log conj "Event 1")
(send log conj "Event 2")
;; Agents process actions in the background
(Thread/sleep 100)
@log  ; Returns ["Event 1" "Event 2"]

;; Var: thread-local binding
(def ^:dynamic *db* {:connection "default"})
(binding [*db* {:connection "test"}]
  (println *db*))  ; Prints {:connection "test"}
(println *db*)     ; Prints {:connection "default"}
```

24.3 Software Transactional Memory

Clojure's refs use Software Transactional Memory (STM) to provide coordinated, atomic updates to multiple references:

```
;; Bank account transfer example
(defn transfer [from to amount]
  (dosync
    (alter from - amount)
    (alter to + amount)))

(def alice (ref 1000))
(def bob (ref 500))

(transfer alice bob 100)
@alice  ; 900
@bob    ; 600

;; Transactions can be retried if there's a conflict
(dotimes [_ 10]
  (future
    (transfer alice bob (rand-int 100))))
```

24.4 Concluding Remarks

Concurrency models in modern programming languages:

- **Go**: Communicating Sequential Processes (CSP) with channels and goroutines
- **Erlang**: Actor model with processes and message passing
- **Clojure**: Software Transactional Memory with functional programming

Each model has its strengths and weaknesses:

- CSP is good for high-throughput, CPU-bound tasks
- Actor model excels in distributed, fault-tolerant systems
- STM works well with shared state in a functional context

The choice of concurrency model depends on the specific requirements of the application:

- Performance characteristics
- Distribution needs
- Fault tolerance requirements
- Programming paradigm preferences

25 Conclusion

This course has covered both the theoretical foundations of concurrency and its practical aspects in modern programming languages:

- **Theory**: Process calculi (CCS, -calculus), bisimulation, fixed point theory, modal logics
- **Practice**: Programming languages (Go, Erlang, Clojure) with built-in concurrency features

The connection between theory and practice is evident in how process calculi have influenced the design of concurrent programming languages:

- CSP $\rightarrow$ Go
- Actor model $\rightarrow$ Erlang
- Functional approaches $\rightarrow$ Clojure

Understanding the theoretical foundations helps in reasoning about concurrent programs and avoiding common pitfalls like race conditions, deadlocks, and livelocks. The practical languages provide efficient and expressive mechanisms for implementing concurrent systems in real-world applications.