

Teaching Concurrency to Freshmen?

Christian Eisentraut Holger Hermanns*
Dependable Systems and Software
Saarland University

*also with INRIA Rhône-Alpes

Context: CS-related degrees offered at Saarland University



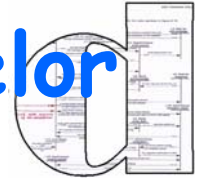
- Computer Science - Bachelor/Master of Science
 - about 150
- Bioinformatics - Bachelor/Master of Science
 - about 20
- Computer and Communication - Bachelor/Master of Science
 - about 40
- Computer Vision - Master of Science
 - less than 10

Context: Freshmen



- ❑ No entrance level requirements
except a general university entrance diploma
(German "Abitur" or equivalent).
- ❑ Basic knowledge in calculus and linear algebra is assumed
- ❑ Department offers a dedicated preparatory course
(discrete math)
 - ❑ 20 lectures and 40 tutorials
 - ❑ spans entire September
 - ❑ precedes the start of the academic year
 - ❑ optional offer, but strongly encouraged

Context: Mandatory courses in CS Bachelor



Math for Computer Scientists 1, 2, 3 (9 CP each)

Programming 1 (9 CP)

Programming 2 (9 CP)

Software Practice Lab (14 CP)

System Architecture (9 CP)

Information Systems (9 CP)

Theoretical Computer Science (9 CP)

ETCS credit points

Each 30 hours of work of the average student is worth one ETCS credit point.

9 CP thus amount to an average workload of 270 hours

Context: Programming Education



Programming 1 (9 CP)

- Foundations of computing
- SML

*Computer science
as executable mathematics*

Programming 2 (9 CP)

- Algorithms and Data Structures
- C/C++

Software Practice Lab (14 CP)

- large software designs
- teams of up to 5 students
- block project during summer break

Programming 1: Original Setup



- ❑ Basic abstract data structures and algorithms
 - ❑ Lists, trees and graphs, list and tree traversal, sorting,....
 - ❑ Practical experiments with their concrete realisations in SML.

 - ❑ Basic considerations of program verification and analysis
 - ❑ Program correctness as integral part of program design
 - ❑ Inductive correctness and termination proofs
 - ❑ Runtime Analysis

 - ❑ Formal syntax and semantics
 - ❑ Abstract and concrete syntax
 - ❑ Lexing and parsing, type checking and
 - ❑ Static and dynamic semantics
 - ❑ Interpreter for a simple subset of SML is developed

 - ❑ Compilation and execution
 - ❑ Programming a virtual machine in SML.
 - ❑ Developing a compiler from an imperative language to VM assembler code.
 - ❑ features: arithmetic operations, jumps, dynamic heap memory allocation, (recursive) procedure calls.
- Success ratio of first years' students is in the order of 70%.
(lower for students who need to retake)*

Basics of Concurrency Theory: What we asked ourselves



- ❏ Which aspects of concurrency theory are of **maximal benefit to the students** for the subsequent courses?
- ❏ How can we **use general concepts already taught** effectively in order to build up a theory of concurrency?
- ❏ Do we need to **sacrifice other parts** of the course we are focusing on?
- ❏ Do we manage to **keep the overall flavor** of the course?

Programming 1: Our twist



- Basic abstract data structures and algorithms
 - Lists, trees and graphs, list and tree traversal, sorting,....
 - Practical experiments with their concrete realisations in SML.
- Basic considerations of program verification and analysis.
 - Program correctness as integral part of program design
 - Inductive correctness and termination proofs
 - Runtime Analysis
- Formal syntax and semantics
 - Abstract and concrete syntax
 - Lexing and parsing, type checking and
 - Static and dynamic semantics

*Success ratio of first years' students
is in the order of 70%.
(lower for students who need to retake)*

Interpreter for a simple subset of SML is developed

Basics of

Compilation and execution

Programming a virtual machine in SML.

Concurrency Theory

Developing a compiler from an imperative language to VM assembler code.

features: an hierarchy of performance, traps, dynamic heap management, call continuation, (non-recursive) procedure calls.

What the students know by then



Basic axiomatic set theory

- ☐ Boolean algebra, relations, functions, recursion, and inductive proofs

Principal concepts of programming languages

- ☐ grammars, type checking, semantics, inference rule and trees

Properties of programs

- ☐ termination and correctness, semantical equivalences of programs, time complexity;

Basic data structures

- ☐ lists, trees, graphs and their representation in SML;

Principles of recursive algorithms

- ☐ list and tree traversals, sorting, divide-and-conquer;

SML basics and some advanced features

- ☐ such as polymorphism

Basics of Concurrency Theory: What we teach



1. We familiarise the students with LTS
 - as a mild extension of directed graphs
2. We introduce a language of processes to generate LTS
 - basically sequential CCS
3. We introduce the composition of processes interacting on complementary signals
 - yielding full CCS
4. We let them implement fragments of syntax and semantics
 - yielding an interactive step-simulator for CCS
5. We elaborate on intriguing concurrency phenomena
 - deadlocks, livelocks, dining philosophers, TODO

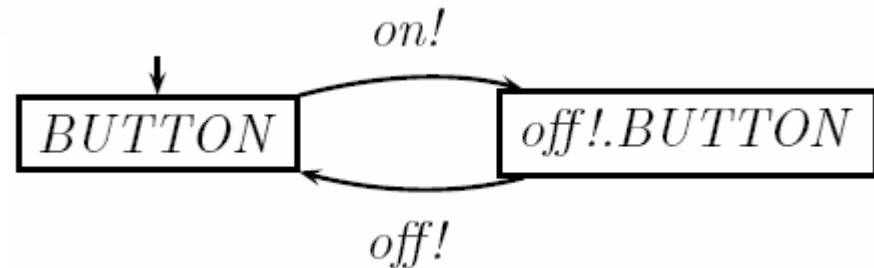
What we teach #1: Labelled transition systems and processes



Definition 1. A (directed) labelled graph G is a triple (V, M, E) with V an arbitrary set of vertices, M an arbitrary set of labels, and $E \subseteq V \times M \times V$ a set of directed labelled edges.

The students are already familiar with graphs.

Definition 2. A process is a pair (G, v) where G is a graph $G = (V, M, E)$ and $v \in V$ is vertex of G .

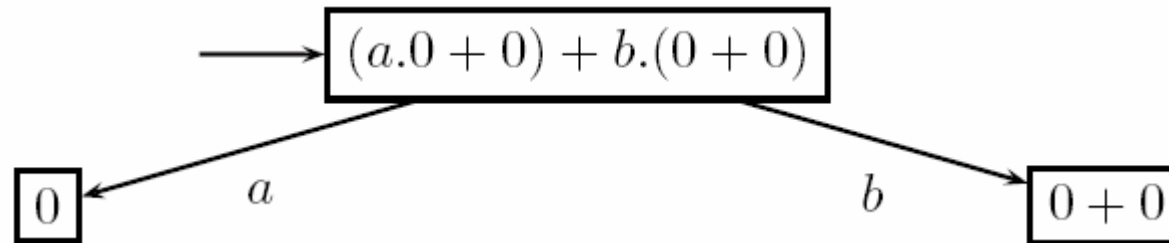


- 🗨 We avoid to emphasize the meaning of labelled graphs and processes as models of concurrency.
- 🗨 Instead we treat them as academic objects.
- 🗨 We only provide hints that a process might be an abstract view on what a computer program does.

What we teach #2: A language for acyclic processes



$$P \in L_0 = 0 \mid a.P \mid P + P \quad \text{where } a \in M$$



with structural operational semantics

$$\llbracket - \rrbracket \in L_0 \rightarrow \mathcal{G}_{L_0} \times L_0 \quad \text{where } \llbracket P \rrbracket = ((L_0, M, \rightarrow), P)$$

$$\mathcal{G}_{L_0} = \{(L_0, M, E) \mid E \subseteq L_0 \times M \times L_0\}$$

where $\rightarrow \subseteq L_0 \times M \times L_0$ is given by the smallest relation satisfying the inference rules

$$\begin{array}{l} \text{prefix} \quad \frac{}{a.P \xrightarrow{a} P} \quad \text{choice}_l \quad \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \quad \text{choice}_r \quad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'} \end{array}$$

What we teach #3: A language for processes



$$P \in L = 0 \quad | \quad a.P \quad | \quad P + P \quad | \quad X$$

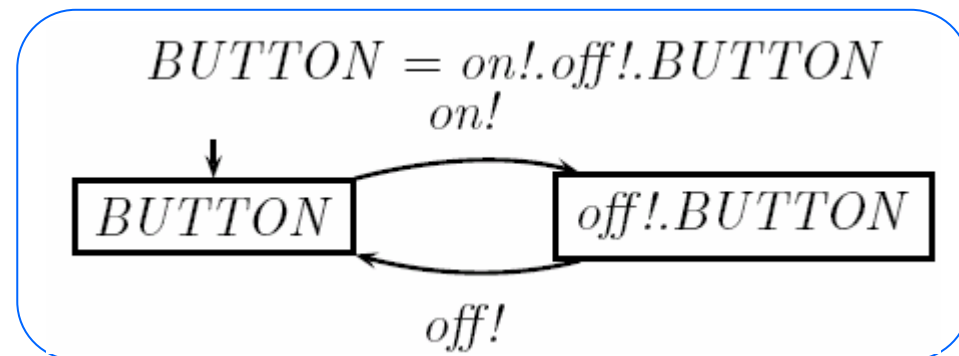
☐ We add a set of defining equations Γ

In Γ all equations have

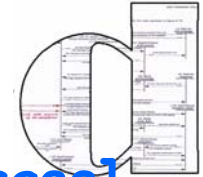
the form $X = E$ where X is a process variable and E is an arbitrary term

☐ and add a rule to deal with recursion

$$\text{rec} \quad \frac{\Gamma(X) = P \quad P \xrightarrow{a} P'}{X \xrightarrow{a} P'}$$



What we teach #4: A cruise control example

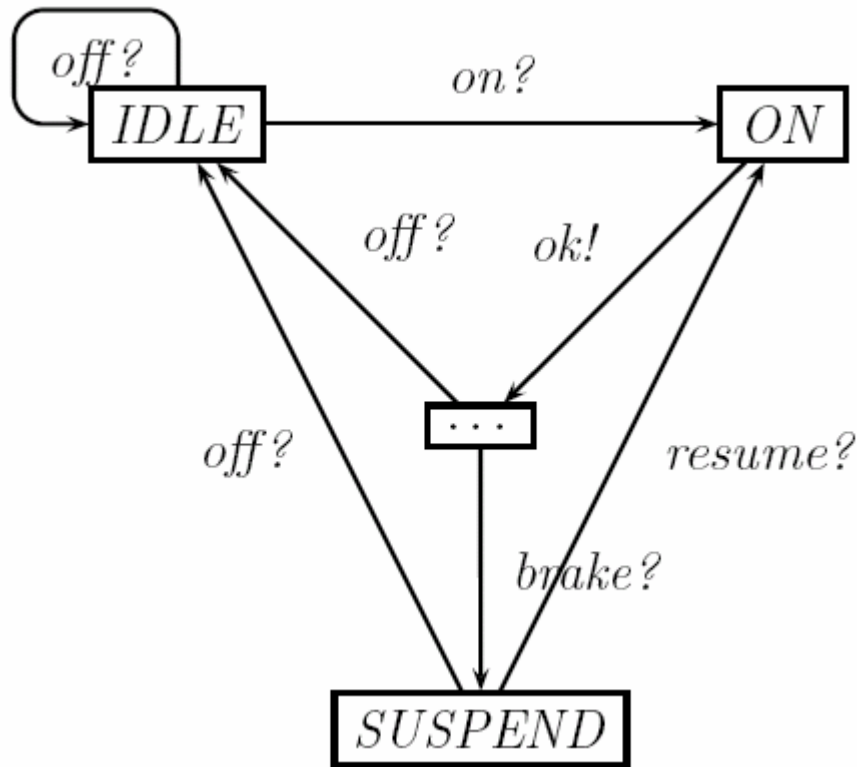


[Kramer/Magee]

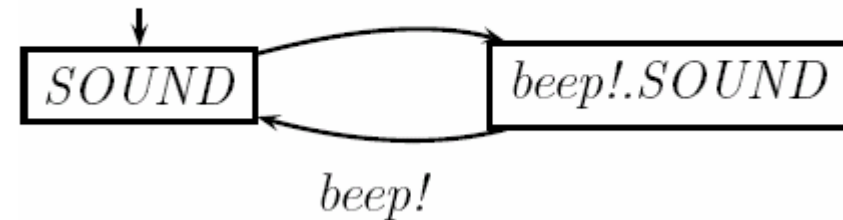
$IDLE = on?.ON + off?.IDLE$

$ON = ok!.(off?.IDLE + brake?.SUSPEND)$

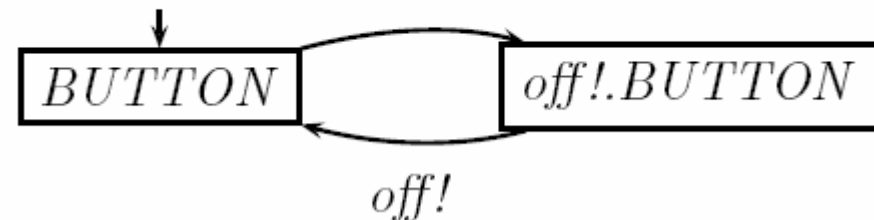
$SUSPEND = resume?.ON + off?.IDLE$



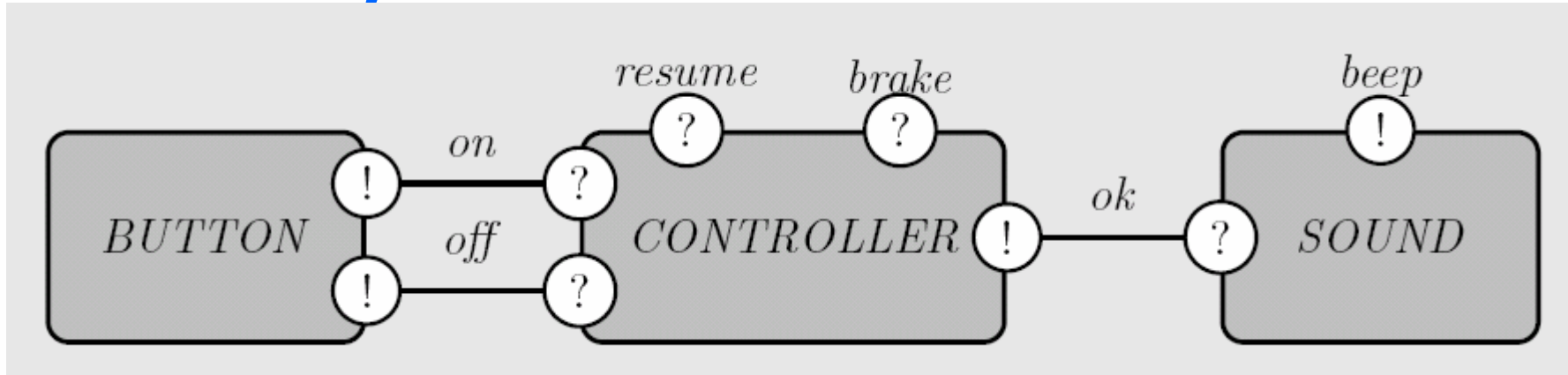
$SOUND = ok?.beep!.SOUND$



$BUTTON = on!.off!.BUTTON$



What we teach #5: Concurrency and interaction



We use this example to postulate the following **principles of concurrency**:

- Real-life processes have states. They can change states via certain actions.
- Actions are atomic, and their purpose is inter-process communication.
- Distinct processes can exist concurrently and perform actions.
- Inter-process communication can be performed whenever pairs of complementary input and output actions occur at the same time.
 - This yields process synchronisation, i.e. a simultaneous change of states.

What we teach #6:

An intermezzo on process synchronisation



We discuss different inter-process communication principles

- Shared variable vs. message passing
- Binary vs. multiway
- Directed vs. undirected
- Buffered vs. handshake

We stay on a very informal level in this intermezzo,
before we return to the above example,
which makes a clear case for
binary, handshaked, directed communication.

What we teach #7: Semantics of CCS parallel operator



We discuss different inter-process communication principles

- Shared variable vs. message passing
- Binary vs. multiway
- Directed vs. undirected
- Buffered vs. handshake

We stay on a very informal level in this intermezzo,
before we return to the above example,
which makes a clear case for (CCS style)
binary, handshaked, directed communication.

$$\text{par}_l \frac{P \xrightarrow{m} P'}{P \mid Q \xrightarrow{m} P' \mid Q} \quad \text{par}_r \frac{P \xrightarrow{m} P'}{Q \mid P \xrightarrow{m} Q \mid P'} \quad \text{sync} \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

What we teach #8: Semantic Equivalence



When should two processes be considered equivalent?

This question re-appeared at various points in our lectures

We reviewed

- graph isomorphism, trace equivalence, and bisimulation equivalence.

The lessons learnt by the students:

- Trace equivalence is the *weakest common criterion*
- The process' *branching structure must be preserved*
so as to avoid deadlocks in the context of parallel composition,
 - isomorphism or bisimulation
- It should be a *congruence for the operators of the language*
 - trace equivalence or bisimulation

In summary, the students understand that for CCS, *bisimulation equivalence* is the central notion of equivalence.

What the students practice #1: They step-simulate CCS examples



CCS> Environment:

X=a!.Z,

Y=((a?.Z + a!.0) + b!.X),

Z=Y

Process:

((X | Y) | Z)

CCS-

CCS> All successors:

```
0.) --a!--> ((Z | Y) | Z)
1.) --a?--> ((X | Z) | Z)
2.) --a!--> ((X | 0) | Z)
3.) --b!--> ((X | X) | Z)
4.) --tau--> ((Z | Z) | Z)
5.) --a?--> ((X | Y) | Z)
6.) --a!--> ((X | Y) | 0)
7.) --b!--> ((X | Y) | X)
8.) --tau--> ((X | 0) | Z)
9.) --tau--> ((X | Z) | 0)
10.) --tau--> ((Z | Y) | Z)
```

CCS- succ 6

CCS> 6-th successor via action a!:

((X | Y) | 0)

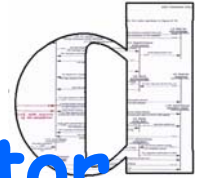
CCS- steps

CCS> All successors:

```
0.) --a!--> ((Z | Y) | 0)
1.) --a?--> ((X | Z) | 0)
2.) --a!--> ((X | 0) | 0)
3.) --b!--> ((X | X) | 0)
4.) --tau--> ((Z | Z) | 0)
```

CCS-

What the students practice #2: They implement fragments of the simulator



Parts of Lexer, Parser, Operational Semantics

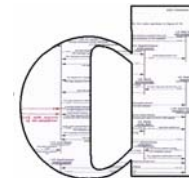
For instance, they were given this

$$\begin{aligned} \text{steps } \Gamma \ 0 &= \emptyset \\ \text{steps } \Gamma \ m.P &= \{(m, P)\} \\ \text{steps } \Gamma \ X &= \text{steps } \Gamma \ (\Gamma X) \\ \text{steps } \Gamma \ (P + Q) &= (\text{steps } \Gamma \ P) \cup (\text{steps } \Gamma \ Q) \\ \text{steps } \Gamma \ (P|Q) &= \{(m, P'|Q) \mid (m, P') \in \text{steps } \Gamma \ P\} \\ &\cup \{(m, P|Q') \mid (m, Q') \in \text{steps } \Gamma \ Q\} \\ &\cup \{(\tau, P'|Q') \mid \exists \alpha : (\alpha, P') \in \text{steps } \Gamma \ P \wedge (\bar{\alpha}, Q') \in \text{steps } \Gamma \ Q\} \end{aligned}$$

and had to complete this

```
(* env -> ccs -> (lab * ccs) list *)
fun steps env Stop = []
  | steps env (Var X) = steps env (env X)
  | steps env (Pre (u,P)) = ...
  | steps env (Chc (P,Q)) = (steps env P) @ (steps env Q)
  | steps env (Par (P,Q)) = (map (fn (a,G) => (a,Par(G,Q)))
                             (steps env P))@...
```

What the students practice #3: They investigate concurrency phenomena



Exercise 136. Consider the following set of recursive equations Γ .

$$Fork1 = getF1?.putF1?.Fork1$$

$$Fork2 = getF2?.putF2?.Fork2$$

$$Fork3 = getF3?.putF3?.Fork3$$

$$PhilA = getF1!.getF2!.eat!.putF1!.putF2!.think!.PhilA$$

$$PhilB = getF2!.getF3!.eat!.putF2!.putF3!.think!.PhilB$$

$$PhilC = getF3!.getF1!.eat!.putF3!.putF1!.think!.PhilC$$

Use *Google* to learn about the “dining philosophers”. Explore

$$Reach(\llbracket (Fork1 \mid PhilA \mid Fork2 \mid PhilB \mid Fork3 \mid PhilC) \rrbracket_{\Gamma}) \setminus H$$

where H contains all actions except $eat!$, $eat?$, $think!$ and $think?$. After which trace will the philosophers have to starve.

with their own implementation!

What is the effect of all this?



- Concurrency basics integrated in very first Bachelor course

- Smooth integration, overall flavor is kept *Computer science as executable mathematics*

- First major programming exercise is about concurrency semantics

- Success ratio unchanged wrt. earlier editions

- Quite succesful

 - Positive evaluation from students

 - Positive feedback from colleagues

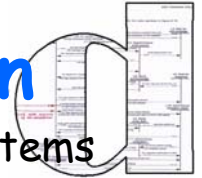
➔ Department decided to revise the Bachelor curriculum, to devote a complete mandatory course to concurrency

Revised CS Bachelor: Mandatory courses



- Math for Computer Scientists 1, 2, 3 (9 CP each)
 - unchanged
- Programming 1 (9 CP)
 - unchanged
- Programming 2 (9 CP)
 - OO and Software Architecture, now Java
- Software Practice Lab (9 instead of 14 CP)
 - streamlined
- System Architecture (9 CP)
 - unchanged
- Information Systems (6 instead of 9 CP)
 - shortened
- Algorithms and Data Structures (6 CP)
 - stripped out of Programming 2
- Concurrent Programming (6 CP)
 - new course
- Theoretical Computer Science (9 CP)
 - unchanged

New mandatory course: Concurrent computation



- ❑ Concurrency as a concept
 - ❑ Potential Parallelism
 - ❑ Actual Parallelism
 - ❑ Conceptual Parallelism
- ❑ Concurrency in practice
 - ❑ Object orientation
 - ❑ Operating Systems
 - ❑ Multi-core processors, coprocessors
 - ❑ Programmed Parallelism
 - ❑ Distributed Systems (client-server, peer-2-peer, data bases, Internet)
 - ❑ Business Processes
- ❑ The Difficulty of Concurrency
 - ❑ Conflicting resources
 - ❑ Fairness
 - ❑ Mutual exclusion
 - ❑ Deadlock, Livelock, Starvation
- ❑ Basics of Concurrency Modelling
 - ❑ Sequential Processes
 - ❑ States, Events, and Transitions
 - ❑ Transition systems
 - ❑ Observable Behaviour
 - ❑ Determinism vs. Non-Determinism
 - ❑ Algebras and Operators
- ❑ CCS: calculus of communicating systems
 - ❑ Sequencing, Choice, Recursion
 - ❑ Concurrency and interaction
 - ❑ Structural operational semantics
 - ❑ Equality of observations
 - ❑ Implementation relations
 - ❑ CCS and data transfer
- ❑ True concurrency models
 - ❑ Petri nets
 - ❑ Partial orders
 - ❑ Event structures
 - ❑ CCS and true concurrency
 - ❑ Other formalisms: MSCs, Statecharts
- ❑ Concurrent Hardware
 - ❑ Transaction Level Modelling
 - ❑ threads, locks, notify, wait.
 - ❑ System C Realisation
- ❑ Programming of Concurrency
 - ❑ Java vs. C++
 - ❑ Objects in Java
 - ❑ Sockets, protocols, and data streams
 - ❑ Shared Objects and Threads in Java
 - ❑ Monitors and Semaphores
- ❑ Analysis and Programming support