[module 1.2]
THE CONCURRENT PROGRAMMING ABSTRACTION
FROM PROGRAMS TO MODELS (AND BACK)

• Importance of *models* and *abstraction* for computer science and engineering in particular
  – rigorous description / representation of program (system) structure and behaviour *at a proper level of abstraction*
    • including relevant information, abstracting from non-relevant aspects
  – diagrammatical representations for program design
  – formal models for program analysis and verification

• Defining proper models for concurrent programs
  – defining models for the structure and behaviour of concurrent programs
    *abstracting from the low-level details of their actual implementation and realization*
    • design
  – enabling the possibility to reason about their dynamic behaviour of concurrent programs
    • verification
CONCURRENT PROGRAMMING MODEL & ABSTRACTION

• Each process is modelled as a sequence of atomic actions, each action corresponding to the atomic execution of an statement.
• The execution of a concurrent program proceeds by executing a sequence of actions obtained by arbitrarily interleaving the actions (atomic statements) from the processes:
  – atomic statements => executed to completion without the possibility of interleaving
  – during the computation the control pointer or instruction of a process indicates the next statement that can be executed by that process.
• A computation or scenario is an execution sequence that can occur as a result of the interleaving.
FIRST TRIVIAL EXAMPLE

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer k1 := 1</td>
<td>p1: n := k1</td>
<td>integer k2 := 2</td>
</tr>
<tr>
<td>integer n := 0</td>
<td></td>
<td>q1: n := k2</td>
</tr>
</tbody>
</table>

• Each labeled line represents an atomic statement
• Each process has private memory
  – local variables, such as k1 and k2
• Processes shares some memory
  – global variables, such as n
STATE DIAGRAMS

• Given the model, the execution of a concurrent program can be formally represented by states and transitions between states
  – the state is defined by a tuple consisting of
    • one element of each process that is a label (statement) from that process
    • one element for each global or local variable that is a value whose type is the same as the type of a variable
  – there is a transition between two states s1 and s2 if executing a statement in state s1 changes the state to s2.
    • the statement executed must be one of those pointed to by a control pointer in s1
• The state diagram is a graph containing all the reachable states of the programs
  – scenarios are represented by directed pathes through the state diagram from the initial state
  – cycles represent the possibility of infinite computation in a finite graph
  – tabular representation
STATE DIAGRAM FOR THE FIRST EXAMPLE

- State tuple: \(<p, q, n, k1, k2>\)

\[
\begin{align*}
<p1, q1, 0, 1, 2> & \\
<end, q1, 1, 1, 2> & \quad <p1, end, 2, 1, 2> \\
<end, end, 2, 1, 2> & \quad <end, end, 1, 1, 2>
\end{align*}
\]
“THE IMPORTANCE OF BEING ATOMIC”

• Atomic increment (1)

<table>
<thead>
<tr>
<th>integer n := 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>p</strong></td>
</tr>
<tr>
<td><strong>q</strong></td>
</tr>
<tr>
<td><strong>p1</strong>: n := n + 1</td>
</tr>
</tbody>
</table>

• Non-atomic increment (2)

<table>
<thead>
<tr>
<th>integer n := 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>p</strong></td>
</tr>
<tr>
<td><strong>q</strong></td>
</tr>
<tr>
<td><strong>p1</strong>: tmp := n</td>
</tr>
<tr>
<td><strong>p2</strong>: n := tmp + 1</td>
</tr>
</tbody>
</table>

• In the second case, a scenario exists in which the final value of n is 1
[NOTE] ASSIGNMENTS & INCREMENTS AT THE MACHINE-CODE LEVEL

- Stack machines

<table>
<thead>
<tr>
<th>integer n := 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
</tr>
<tr>
<td>p1: push n</td>
</tr>
<tr>
<td>p2: push #1</td>
</tr>
<tr>
<td>p3: add</td>
</tr>
<tr>
<td>p4: pop n</td>
</tr>
</tbody>
</table>

- Register machines

<table>
<thead>
<tr>
<th>integer n := 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
</tr>
<tr>
<td>p1: load R1, n</td>
</tr>
<tr>
<td>p2: add R1,#n</td>
</tr>
<tr>
<td>p3: store n, R1</td>
</tr>
</tbody>
</table>
[NOTE] NON-ATOMIC VARIABLES (1/2)

• The notion of “atomic” can be referred not only to actions, but also to data structures:
  – a data object is defined *atomic* if it can be in a finite number of states equals to the number of values that it can assume
    • operations change (atomically) that state
  – typically primitive data type in concurrent languages are atomic
    • not always: e.g. *double* in Java

• Abstract data types composed by multiple simpler data objects are typically non atomic
  – *es*: class in OO languages, structs in C

• In that case for the ADT (or more generally data object) it is possible to identify two basic types of states: *internal* and *external*
  – the internal state is meaningful for who defines the data object (class)
  – the external state is meaningful for who uses the data object

• The correspondence among internal and external states is *partial*
  – there exist internal states which have no a correspondent external state
  – internal states which have a correspondent external state are defined *consistent*
[NOTE] NON-ATOMIC VARIABLES (2/2)

- Then, the execution of an operation on a (not-atomic) ADT can go through states that are *not consistent*
  - E.g. a simple list
- This is not a problem in the case of sequential programming
  - thanks to information hiding
- Conversely, it is a problem in the case of concurrent programming
  - it can happen that a process would work on an object while the object is in an inconsistent state, since an process is concurrently operating on it

> it is necessary to introduce proper mechanisms that would guarantee that processes work on data objects that are always in states that are consistent
STATE DIAGRAM OF CYCLIC PROCESSES

- **p** and **q** processes cycling on a condition

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
<td>while (n &lt; 1)</td>
<td>q1: while n &gt;= 0</td>
</tr>
<tr>
<td>p2:</td>
<td>n := n + 1</td>
<td>q2: n := n - 1</td>
</tr>
</tbody>
</table>

integer n := 1

- **Exercises**
  - state diagram ?
  - construct a scenario in which the loop in p executes exactly one
  - construct a scenario in which the loop in p executes exactly three times
  - construct a scenario in which both loops execute infinitely often
AN EXAMPLE WITH N PROCESSES

• N processes with the same program, indexed by index i in [0..N-1]

| Integer array[0..N-1] vect1 := \{initialized with some values\} 
| Integer array[0..N-1] vect2 

<table>
<thead>
<tr>
<th>p[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer myNum, count</td>
</tr>
</tbody>
</table>

| p1: myNum := vect1[i] |
| p2: count := \<number of elements of vect1 less than myNum\> |
| p3: vect2[count] := myNum |

• What the algorithm do?
STATE DIAGRAM OF
NON INTERACTING PROCESSES

• P,Q processes composed by \{p_1,p_2,p_3,...\} and \{q_1,q_2,q_3,...\} fully independent statements
IS THIS MODEL A GOOD MODEL?
THAT IS: IS THE CONCURRENT PROGRAMMING ABSTRACTION JUSTIFIABLE?

- Actually in the reality computer system **has not a global state**
  - matter of physics
- That's the role of abstraction: **we create a model of the system in which a kind of global entity executes the concurrent program by arbitrarily interleaving statements**
  - to ease analysis
- Is it a valid model for real concurrent computing systems? Reality check
  - multitasking systems
  - multicore systems
  - multiprocessor computers
  - distributed systems
ARBITRARILY INTERLEAVING: ABSTRACTING FROM TIME

- Arbitrary interleaving means that we ignore time in our analysis of concurrent programs
  - focusing only to
    - partial orders related to action sequences a1,a2,...
    - atomicity of the individual action aj => choosing what is atomic is fundamental
  - robustness w.r.t. both hardware (processor) and software changes
    - independent from changes in timings / performance
- This makes concurrent programs amenable to formal analysis, which is necessary to ensure correctness of concurrent programs.
  - proving correctness besides the actual execution time, which is typically strictly dependent on processors speed and system's environment timings
CORRECTNESS OF PROGRAMS

• Checking correctness for sequential programs
  – unit testing based on specified input and expecting some specified output
    • diagnose, fix, rerun cycle
  – re-running a program with the same input will always give the same result

• Concurrent programming new (challenging) perspective
  – the same input can give different outputs (depending on the scenario...)
    • some scenarios may give correct output while others do not
  – you cannot debug a concurrent program in the normal way because each time you run the program, you will likely get a different scenario

• Needs of different kind of approaches
  – formal analysis, model checking
  – based on abstract models
CORRECTNESS OF CONCURRENT PROGRAMS

- The correctness of (possibly non-terminating) concurrent programs is defined in terms of properties of computations
  - condition (assertion) that must be verified in every possible scenarios
- Two type of correctness properties
  - safety property
  - liveness property
SAFETY PROPERTIES

- The property must be **always** true, i.e. for a safety property $P$ to hold, it must be true in every state of every computation
  - expressed as invariants of a computation
- Typically used to specify that “bad things” should never happen
  - mutual exclusion
    - no more than one process is ever present in a critical region
  - no deadlock
    - no process is ever delayed awaiting an event that cannot occur
  - ...
LIVENESS (OR PROGRESS) PROPERTY

• The property must **eventually** become true
  – i.e. for a liveness property $P$ to hold, it must be true that in every computation there is some state in which $P$ is true
• Typically used to specify that “good things” eventually happen
  – no starvation
    • a process finally gets the resource it needs (CPU time, lock)
  – no dormancy
    • a waiting process is finally awakened
  – reliable communication
    • a message sent by one process to another will be received
  – ...
• **Fairness**
  – a liveness property which holds that something good happens infinitely often
    • ex: a process activated infinitely often during an application execution, each process getting a fair turn
WEAKLY FAIR SCENARIO

• def. weakly fair scenario
  – a scenario is (weakly) fair if at any state in the scenario a statement that is continually enabled eventually appears in the scenario

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
<td>while flag = false</td>
<td>q1: flag := true</td>
</tr>
<tr>
<td>p2:</td>
<td>n := 1 - n</td>
<td></td>
</tr>
</tbody>
</table>

• Does this algorithm necessarily halt?
• The non-terminating scenario is not fair
  – if we allow only for fair scenario, then eventually an execution of q1 must be included in every scenario
SOME EXERCISES (1/2)

integer n := 0

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer temp</td>
<td>integer temp</td>
</tr>
<tr>
<td>p1: do 10 times</td>
<td>q1: do 10 times</td>
</tr>
<tr>
<td>p2: temp := n</td>
<td>q2: temp := n</td>
</tr>
<tr>
<td>p3: n := temp + 1</td>
<td>q3: n := temp + 1</td>
</tr>
</tbody>
</table>

Construct a scenario in which the final value is 2

integer n := 0

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: while n &lt; 2</td>
<td>q1: n := n + 1</td>
</tr>
<tr>
<td>p2: write(n)</td>
<td>q2: n := n + 1</td>
</tr>
</tbody>
</table>

Draw the state diagram

Construct scenarios that give the output sequences: 012, 002, 012

Must the value 2 appear in the output? How many times can 2 appear in the output? How many times can 1 appear in the output?
SOME EXERCISES (2/2)

- **Welfare crook** problem
  - let a, b, c be three ordered array of integer elements. It is known that some element appears in each of the three array. Here it is an outline of a sequential algorithm to find the smallest indices i, j, k, for which a[i] = b[j] = c[k]

```
integer array[0..N] a, b, c := < as required >
integer i := 0, j := 0, k := 0

loop
  p1: if condition-1
  p2:   i := i + 1
  p3: else if condition-2
  p4:   j := j + 1
  p5: else if condition-3
  p6:   k := k + 1
        else exit loop

  write conditional expressions that make the algorithm correct
  develop a concurrent algorithm for this problem
```