v1.0

Sistemi Concorrenti e di Rete LS

II Facoltà di Ingegneria - Cesena
a.a 2008/2009

[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

`integer n := 0`	
p	**q**
`integer k1 := 1`	`integer k2 := 2`
p1: `n := k1`	**q1:** `n := k2`

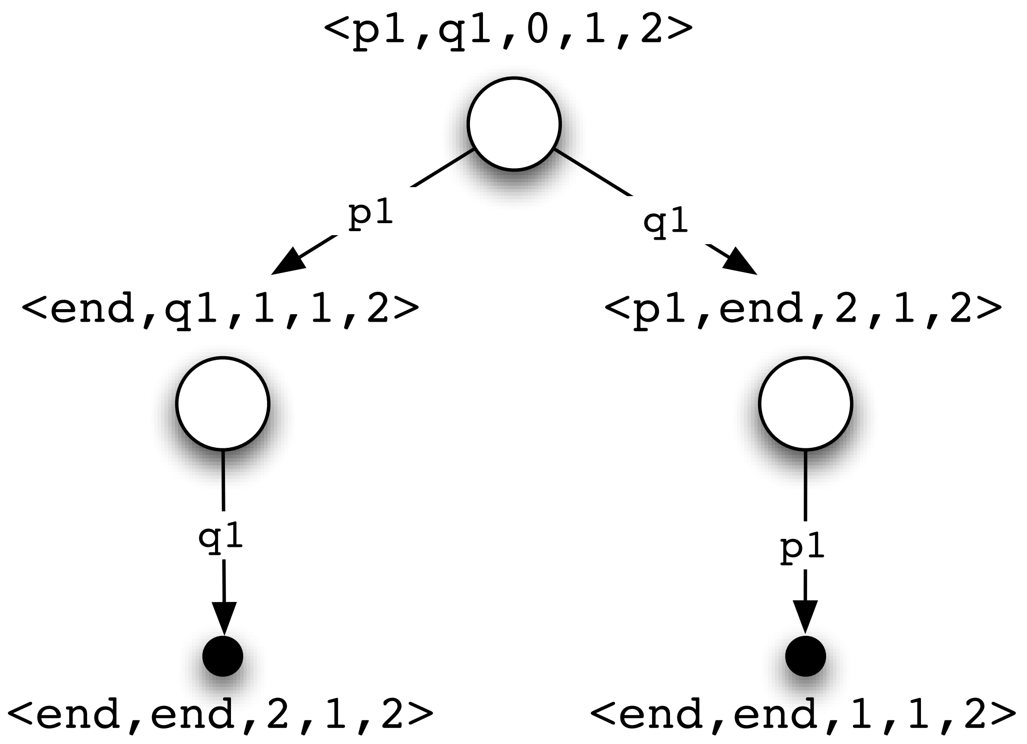
- Each labeled line represents an atomic statement
- Each process has private memory
  - local variables, such as k1 and k2
- Processes shares some memory
  - global variables, such as n

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>`



`<p1,q1,0,1,2>`

p1

q1

`<end,q1,1,1,2>`

`<p1,end,2,1,2>`

q1

p1

`<end,end,2,1,2>`

`<end,end,1,1,2>`

“THE IMPORTANCE OF BEING ATOMIC”

- Atomic increment (1)

`integer n := 0`	
p	**q**
`p1: n := n + 1`	`q1: n := n + 1`

- Non-atomic increment (2)

`integer n := 0`	
p	**q**

`integer tmp;`
`p1: tmp := n`
`p2: n := tmp + 1`
	`integer tmp;`
	`q1: tmp := n`
	`q2: n := tmp + 1`

- 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

`integer n := 0`	
p	**q**
p1: `push n` **p2:** `push #1` **p3:** `add` **p4:** `pop n`	**q1:** `push n` **q2:** `push #1` **q3:** `add` **q4:** `pop n`

- Register machines

`integer n := 0`	
p	**q**
p1: `load R1, n` **p2:** `add R1,#n` **p3:** `store n, R1`	**q1:** `load R1, n` **q2:** `add R1,#n` **q3:** `store n, R1`

[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

```
integer n := 1
```

p

```
p1: while (n < 1)
p2:   n := n + 1
```

q

```
q1: while n >= 0
q2:   n := n - 1
```

- Exercises
  - state diagram ?
  - construct a scenario in which the loop in p executes exaclty 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

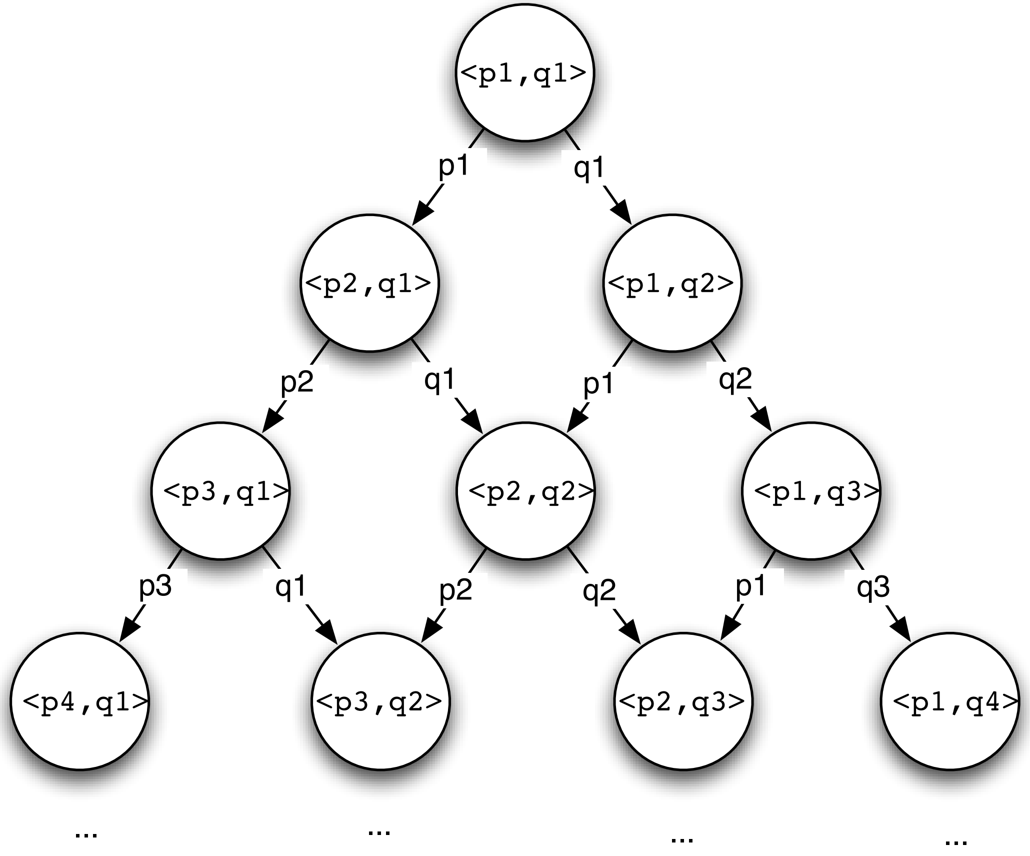
- N processes with the same program, indexed by index i in [0..N-1]

```
integer array[0..N-1] vect1 := {initialized with some values }
integer array[0..N-1] vect2

                              p[i]

  integer myNum, count

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 {p1,p2,p3,...} and {q1,q2,q3,...} 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 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 ingore time in our analysis of concurrent programs
  - focussing only to
    - partial orders related to action sequences a1,a2,...
    - atomicity of the individual action aj => chosing what is atomic is fundamental
  - robustness w.r.t. both hardware (processor) and software changes
    - indepedent 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 envirorонment 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 computationsì
- 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

```
integer n := 0
boolean flag := false
```

p

```
p1: while flag = false
p2:   n := 1 - n
```

q

```
q1: flag := true
```

- 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
```

p

```
    integer temp
p1: do 10 times
p2:   temp := n
p3:   n := temp + 1
```

q

```
    integer temp
q1: do 10 times
q2:   temp := n
q3:   n := temp + 1
```

- Construct a scenario in which the final value is 2

```
integer n := 0
```

p

```
p1: while n < 2
p2:   write(n)
```

q

```
q1: n := n + 1
q2: n := n + 1
```

- 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