[module 2.2]
MODELING CONCURRENT PROGRAM EXECUTION
SUMMARY

• Making models of concurrent programs
• State diagrams & execution scenarios
• Correctness
  – safety and liveness properties
  – fairness
FROM PROGRAMS TO MODELS (AND BACK)

• Importance of **models** and **abstraction** for computer science and engineering in particular
  – model: rigorous description / representation of program (system) structure and behavior *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 behavior of concurrent programs *abstracting from the low-level details of their actual implementation and realization*
    • design
  – enabling the possibility to reason about their dynamic behavior of concurrent programs
    • verification
A MODEL FOR CONCURRENT PROGRAM EXECUTION

• Modeling each process as a sequence of atomic actions, each action corresponding to the atomic execution of a statement

• Speed independence assumption => modeling the execution of a concurrent program as a sequence of actions obtained by arbitrarily interleaving the actions (atomic statements) from the processes
  – a single abstract global processor executing all the actions
  – 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>integer n := 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
</tr>
<tr>
<td>integer k1 := 1</td>
</tr>
<tr>
<td>p1: n := k1</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
- Program execution: 2 scenarios
  - p1, q1 (=> n finally is equal to 2)
  - q1, p1 (=> n finally is equal to 1)
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 $s_1$ and $s_2$ if executing a statement in state $s_1$ changes the state to $s_2$.
    • the statement executed must be one of those pointed to by a control pointer in $s_1$
• The **state diagram** is a **graph** containing all the **reachable states** of the programs
  – scenarios are represented by directed paths through the state diagram from the initial state
  – cycles represent the possibility of infinite computation in a finite graph
STATE DIAGRAM FOR THE FIRST EXAMPLE

• State tuple: \(<p, q, n, k1, k2>\)

• 5 states, 2 scenarios
EXAMPLE #2

- State diagram?
- How many scenarios?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>p</strong></td>
<td><strong>q</strong></td>
</tr>
<tr>
<td>p1: print(&quot;p1&quot;)</td>
<td>q1: print(&quot;q1&quot;)</td>
</tr>
<tr>
<td>p2: print(&quot;p2&quot;)</td>
<td>q2: print(&quot;q2&quot;)</td>
</tr>
</tbody>
</table>
EXAMPLE #2

- 6 scenarios:

\[
\begin{align*}
\text{p1 p2 q1 q2} \\
\text{p1 q1 p2 q2} \\
\text{p1 q1 q2 p1} \\
\text{q1 q2 p1 p2} \\
\text{q1 p1 q2 p2} \\
\text{q1 p1 p2 q2}
\end{align*}
\]
### EXAMPLE #3

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: print(“p1”)</td>
<td>q1: print(“q1”)</td>
</tr>
<tr>
<td>p2: print(“p2”)</td>
<td>q2: print(“q2”)</td>
</tr>
<tr>
<td>p3: print(“p3”)</td>
<td>q3: print(“q3”)</td>
</tr>
</tbody>
</table>

- Scenarios?
EXAMPLE #3

- 20 scenarios:

```
p1  p2  p3  q1  q2  q3
p1  p2  q1  p3  q2  q3
p1  q1  p2  p3  q2  q3
q1  p1  p1  p3  q2  q3
p1  p2  q1  q2  p3  q3
p1  q1  p2  p2  p3  q3
p1  q1  q2  p2  p3  q3
q1  p1  q2  p2  p3  q3
q1  q2  p1  p2  p3  q3
q1  q1  q2  q3  p2  p3
q1  p1  q2  q3  p2  p3
q1  q2  p1  q3  p2  p3
q1  q2  q3  p1  p2  p3
```
### NUM. SCENARIOS WITH 2 PROCESSES

<table>
<thead>
<tr>
<th>num. actions per proc.</th>
<th>num. scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>70</td>
</tr>
<tr>
<td>5</td>
<td>252</td>
</tr>
<tr>
<td>6</td>
<td>924</td>
</tr>
<tr>
<td>7</td>
<td>3432</td>
</tr>
<tr>
<td>8</td>
<td>12820</td>
</tr>
<tr>
<td>...</td>
<td>....</td>
</tr>
</tbody>
</table>

- Note: in these cases process actions have no dependencies...
EXAMPLE #4

- Changing the number of processes: 3 processes.

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
<td>print(“p1”)</td>
<td>q1:</td>
<td>print(“q1”)</td>
</tr>
<tr>
<td>p2:</td>
<td>print(“p2”)</td>
<td>q2:</td>
<td>print(“q2”)</td>
</tr>
</tbody>
</table>

- Scenarios?
**EXAMPLE #4**

- 90 scenarios

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>q1</th>
<th>q2</th>
<th>r1</th>
<th>r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
<td>q1</td>
<td>r1</td>
<td>q2</td>
<td>r2</td>
</tr>
<tr>
<td>p1</td>
<td>p2</td>
<td>q1</td>
<td>r1</td>
<td>r2</td>
<td>q2</td>
</tr>
<tr>
<td>p1</td>
<td>p2</td>
<td>r1</td>
<td>q1</td>
<td>q2</td>
<td>r2</td>
</tr>
<tr>
<td>p1</td>
<td>p2</td>
<td>r1</td>
<td>q1</td>
<td>r2</td>
<td>q2</td>
</tr>
<tr>
<td>p1</td>
<td>p2</td>
<td>r1</td>
<td>r2</td>
<td>q1</td>
<td>q2</td>
</tr>
<tr>
<td>p1</td>
<td>q1</td>
<td>p2</td>
<td>q2</td>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q1</td>
<td>p1</td>
<td>p2</td>
<td>q2</td>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>q1</td>
<td>p1</td>
<td>p2</td>
<td>r1</td>
<td>q2</td>
<td>r2</td>
</tr>
<tr>
<td>q1</td>
<td>p1</td>
<td>p2</td>
<td>r1</td>
<td>r2</td>
<td>q2</td>
</tr>
<tr>
<td>q1</td>
<td>p1</td>
<td>q2</td>
<td>p2</td>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>q1</td>
<td>p1</td>
<td>q2</td>
<td>r1</td>
<td>p2</td>
<td>r2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td>p1</td>
<td>p2</td>
<td>q1</td>
<td>q2</td>
<td>r2</td>
</tr>
<tr>
<td>r1</td>
<td>p1</td>
<td>p2</td>
<td>q1</td>
<td>r2</td>
<td>q2</td>
</tr>
<tr>
<td>r1</td>
<td>p1</td>
<td>p2</td>
<td>r2</td>
<td>q1</td>
<td>q2</td>
</tr>
<tr>
<td>r1</td>
<td>p1</td>
<td>q1</td>
<td>p2</td>
<td>q2</td>
<td>r2</td>
</tr>
<tr>
<td>r1</td>
<td>p1</td>
<td>q1</td>
<td>p2</td>
<td>r2</td>
<td>q2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td>r2</td>
<td>q1</td>
<td>p1</td>
<td>q2</td>
<td>p2</td>
</tr>
<tr>
<td>r1</td>
<td>r2</td>
<td>q1</td>
<td>q2</td>
<td>p1</td>
<td>p2</td>
</tr>
</tbody>
</table>
### NUM. SCENARIOS WITH 3 PROCESSES

<table>
<thead>
<tr>
<th>num. actions per proc.</th>
<th>num. scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>90</td>
</tr>
<tr>
<td>3</td>
<td>1680</td>
</tr>
<tr>
<td>4</td>
<td>34650</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
GENERALIZING...

- Number of scenarios produced by $n$ processes, each having $m_i$ actions:

\[ ns = \frac{(\sum_{i=1}^{n} m_i)!}{\prod_{i=1}^{n} (m_i)!} \]

<table>
<thead>
<tr>
<th>$m_i$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>90</td>
<td>2520</td>
<td>113400</td>
<td>$2^{22.8}$</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>1680</td>
<td>$2^{18.4}$</td>
<td>$2^{27.3}$</td>
<td>$2^{36.9}$</td>
</tr>
<tr>
<td>4</td>
<td>70</td>
<td>34650</td>
<td>$2^{25.9}$</td>
<td>$2^{38.1}$</td>
<td>$2^{51.5}$</td>
</tr>
<tr>
<td>5</td>
<td>252</td>
<td>$2^{19.5}$</td>
<td>$2^{33.4}$</td>
<td>$2^{49.1}$</td>
<td>$2^{66.2}$</td>
</tr>
<tr>
<td>6</td>
<td>924</td>
<td>$2^{24.0}$</td>
<td>$2^{41.0}$</td>
<td>$2^{60.2}$</td>
<td>$2^{81.1}$</td>
</tr>
</tbody>
</table>
“THE IMPORTANCE OF BEING ATOMIC”

- Atomic increment (1)

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
<td>n := n + 1</td>
<td>q1: n := n + 1</td>
</tr>
</tbody>
</table>

- Non-atomic increment (2)

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
<td>tmp := n</td>
<td>q1: tmp := n</td>
</tr>
<tr>
<td>p2:</td>
<td>n := tmp + 1</td>
<td>q2: n := tmp + 1</td>
</tr>
</tbody>
</table>

- In the second case, scenarios exist in which the final value of n is 1
NON ATOMIC CASE: STATE DIAGRAM

12 states
Only 2 scenarios over 6 have final n value = 2
ASSIGNMENTS & INCREMENTS AT THE MACHINE-CODE LEVEL

- **Stack machines**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: push n</td>
<td>q1: push n</td>
</tr>
<tr>
<td>p2: push #1</td>
<td>q2: push #1</td>
</tr>
<tr>
<td>p3: add</td>
<td>q3: add</td>
</tr>
<tr>
<td>p4: pop n</td>
<td>q4: pop n</td>
</tr>
</tbody>
</table>

- **Register machines**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: load R1, n</td>
<td>q1: load R1, n</td>
</tr>
<tr>
<td>p2: add R1,#n</td>
<td>q2: add R1,#n</td>
</tr>
<tr>
<td>p3: store n, R1</td>
<td>q3: store n, R1</td>
</tr>
</tbody>
</table>
NON-ATOMIC STRUCTURES (1/3)

• 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
NON-ATOMIC STRUCTURES (2/3)

- 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*
NON-ATOMIC STRUCTURES (3/3)

• 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
CYCLIC PROCESSES

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

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

**Analysis**

- 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

integer n := 1
IS THIS MODEL A GOOD MODEL?
OR RATHER: 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
• But.... 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
  - focussing 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
  > we can’t debug a concurrent program in the normal way
    • each time you run the program we 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
  – conditions 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 computations
• 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
- Main example
  - a process activated infinitely often during an application execution, each process getting a fair turn
  - i.e. an action that can be executed, eventually will be executed
    - requirement on the scheduling
- So programs can have different liveness behavior depending on precisely how their instructions are interleaved
  - how instructions are interleaved is a result of a scheduling policy.
FAIRNESS & SCHEDULING POLICIES

• Unconditional Fairness
  – a scheduling policy is unconditionally fair if every unconditional atomic action that is eligible is executed eventually

• Weak Fairness
  – a scheduling policy is weakly fair if it is unconditionally fair and every eligible conditional atomic action whose condition becomes and remains true is executed eventually.

• Strong Fairness
  – a scheduling policy is strongly fair if it is unconditionally fair and every eligible conditional atomic action whose condition becomes true infinitely often (infinitely many times) is executed eventually.
UNCONDITIONALLY FAIR SCENARIO

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

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

integer n := 0
boolean flag := false

• Does this algorithm necessarily halt?
  – yes if we assume only fair scenarios
    • if we allow only fair scenario, then eventually an execution of q1 must be included in every scenario
  – the non-terminating scenario is not fair
SUMMARY

• Making models of concurrent programs
• State diagrams & execution scenarios
• Correctness
  – safety and liveness properties
  – fairness