Concurrency at Programming Language Level

Ch 2 [BenA 06]

Abstraction
Pseudo-language
BACI
Ada, Java, etc.
Levels of Abstraction

- **Granularity of operations**
  - Invoke a library module
  - Statement in high level programming language
  - Instruction in machine language

- **Atomic statement**
  - Anything that we can guarantee to be atomic
    - Executed completely “at once”
    - Always the same correct atomic result
    - Result does not depend on anybody else
  - Can be at any granularity
  - Can *trust* on that atomicity
Atomic Statement

- Atomicity guaranteed somehow
  - Machine instruction: HW
    - Memory bus transaction
  - Programming language statement, set of statements, or set of machine instructions
    - SW
      - Manually coded
      - Disable interrupts
      - OS synchronization primitives
  - Library module
    - SW
      - Manually coded inside
      - Provided automatically to the user by programming environment

Load R1, Y
Read mem(0x35FA8300)
-- start atomic
Load R1, Y
Sub R1, =1
Jpos R1, Here
-- end atomic

Monitors
Ch 7 [BenA 06]
Concurrent Program

- Sequential process
  - Successive atomic statements
    
    \[ P: p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \ldots \]
  - Control pointer (\(=\) program counter)

- Concurrent program
  - Finite set of sequential processes working for same goal
  - Arbitrary interleaving of atomic statements in different processes

3 processes (P, R, Q) interleaved execution

P: p_1 \rightarrow p_2
Q: q_1 \rightarrow q_2

\[ p_1, r_1, p_2, q_1 \]
\[ q_2, \ldots \]
\[ \uparrow c_{p_R} \]
\[ p_3, \ldots \]
\[ \uparrow c_{p_P} \]
\[ r_2, \ldots \]
\[ \uparrow c_{p_Q} \]

\[ p_1 \rightarrow q_1 \rightarrow p_2 \rightarrow q_2, \]
\[ p_1 \rightarrow q_1 \rightarrow q_2 \rightarrow p_2, \]
\[ p_1 \rightarrow p_2 \rightarrow q_1 \rightarrow q_2, \]
\[ q_1 \rightarrow p_1 \rightarrow q_2 \rightarrow p_2, \]
\[ q_1 \rightarrow p_1 \rightarrow p_2 \rightarrow q_2, \]
\[ q_1 \rightarrow q_2 \rightarrow p_1 \rightarrow p_2. \]

P: p_1 \rightarrow p_2
Q: q_1 \rightarrow q_2

\[ p_1 \rightarrow q_2 \rightarrow p_2 \rightarrow q_1 \]
Program State, Pseudo-language

- **Sequential program**

- **State**
  - next statement to execute (cp, i.e., PC)
  - variable values
(Global) Program State

- Concurrent program

<table>
<thead>
<tr>
<th>Algorithm 2.1: Trivial concurrent program</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer n ← 0</td>
</tr>
</tbody>
</table>

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

- **Local state for each process:**
  - cp
  - Variable values
    - Local & global

- **Global state for program**
  - All cp’s
  - All local variables
  - All global variables
Possible Program States

- List of processes in program
  - List of values for each process
    - \( cp \)
    - \( \text{value of each local/global/shared variable} \)

\[ \text{state:} \{ \{ p_1: n \leftarrow k_1 \} \quad \text{process p} \]
\[ k_1 = 1 \}
\[ \{ q_1: n \leftarrow k_2 \} \quad \text{process q} \]
\[ k_2 = 2 \}
\[ n = 0 \quad \text{shared variable} \}

- \( \text{Nr of possible states} \)
  - \( \text{can be (very) large} \)
  - \( \text{Not all states are reachable states!} \)
  - \( \text{Different executions do not go through same states (even with same input)} \)

\[ \text{unreachable state:} \{ \} \]

\( k_1 = 1, k_2 = 2 \)
\( n = 0 \)
- Transitions from one possible state to another
  - Executed statement must be one of those in the 1st state
- State diagram for concurrent program
  - Contains all reachable states and transitions
  - All possible executions are included, they are all correct!
Atomic Statements

- Two scenarios
  - Both correct
  - Different result!

NO need to have the same result! Statements do the same, but overall result may be different. (see p. 19 [BenA 06])

- Atomic?
  - Assignment?
  - Boolean evaluation?
  - Increment?

18.1.2011
Algorithm 2.3: Atomic assignment statements

\[
\text{integer } n \leftarrow 0
\]

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
<td>(n \leftarrow n + 1)</td>
<td>q1: (n \leftarrow n + 1)</td>
</tr>
</tbody>
</table>

- Two scenarios for execution
  - Both correct
  - Both have the same result

<table>
<thead>
<tr>
<th>Process p</th>
<th>Process q</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p1: n \leftarrow n + 1)</td>
<td>(q1: n \leftarrow n + 1)</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Process p} & \quad \text{Process q} & \quad n \\
p1: n \leftarrow n + 1 & \quad q1: n \leftarrow n + 1 & \quad 0 \\
\text{(end)} & \quad \text{(end)} & \quad 1 \\
\text{(end)} & \quad \text{(end)} & \quad 2
\end{align*}
\]

<table>
<thead>
<tr>
<th>Process p</th>
<th>Process q</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p1: n \leftarrow n + 1)</td>
<td>(q1: n \leftarrow n + 1)</td>
<td>0</td>
</tr>
<tr>
<td>(p1: n \leftarrow n + 1)</td>
<td>(end)</td>
<td>1</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>2</td>
</tr>
</tbody>
</table>
Algorithm 2.3: Atomic assignment statements

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
<td>\textcolor{red}{n \leftarrow n + 1}</td>
<td>q1:</td>
</tr>
</tbody>
</table>

Same statements with smaller atomic granularity:

Algorithm 2.4: Assignment statements with one global reference

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
<td>\textcolor{red}{\text{integer temp}} \textcolor{red}{\text{temp} \leftarrow n}</td>
<td>q1:</td>
</tr>
<tr>
<td>p2:</td>
<td>\textcolor{red}{\text{integer temp}} \textcolor{red}{n \leftarrow \text{temp + 1}}</td>
<td>q2:</td>
</tr>
</tbody>
</table>
Too Small Atomic Granularity

- Scenario 1
  - OK
- Scenario 2
  - Bad result
- From now on
  - Assignments and Boolean evaluations are atomic!
Correctness

• What is the correct answer?
• Usually clear for sequential programs
• Can be fuzzy for concurrent programs
  – Many correct answers?
  – What is intended semantics of the program?
  – Run programs 100 times, each time get different answer?
    • Each answer is correct, if program is correct!
    • Does not make debugging easier!
    • Usually can not test all possible scenarios (too many!)
  – How to define correctness for concurrent programs?
    • Safety properties = properties that are always true
    • Liveness properties = properties that eventually become true
Safety and Liveness

- **Safety property**
  - property must be true all the time ("bad" never happens)
    - “Identity”
      - memFree + memAllocated = memTotal
    - Mouse cursor is always displayed
    - System responds always to new commands

- **Liveness property**
  - Property must eventually become true ("good" eventually happens)
    - Variable n value = 2
    - System prompt for next command is shown
    - Control will resume to calling program
    - Philosopher will get his turn to eat
    - Eventually the mouse cursor is not displayed
    - Program will terminate

- **Duality of safety and liveness properties**
  - { P_i will get his turn to eat } \equiv \neg \{ P_i will never get his turn to eat \}
  - { n value will become 2 } \equiv \neg \{ n value is always \neq 2 \}
Linear Temporal Logic (LTL)

- Define safety and liveness properties for certain state in some (arbitrary) scenario
  - Example of Modal Temporal Logic (MDL), logic on concepts like possibility, impossibility, and necessity

- Alternative: Branching Temporal Logic (BTL)
  - Properties true in some or all states starting from the given state
    - More complex, because all future states must be covered
  - Common Temporal Logic (CTL)
    - Can be checked automatically
      - Every time computation reaches given state
    - SMV model checker
    - NuSMV model checker
Fairness

- (Weakly) fair scenario
  - **Wanted condition** eventually occurs
    - Nobody is locked out forever?
    - Will a philosopher ever get his turn to eat?
    - Will an algorithm eventually stop?
    - $p$ and $q$ are both scheduled to run eventually

<table>
<thead>
<tr>
<th>Algorithm 2.5: Stop the loop A</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer $n \leftarrow 0$</td>
</tr>
<tr>
<td>boolean flag $\leftarrow$ false</td>
</tr>
<tr>
<td>$p$</td>
</tr>
<tr>
<td>$p1$: while flag = false</td>
</tr>
<tr>
<td>$p2$: $n \leftarrow 1 - n$</td>
</tr>
</tbody>
</table>

- All scenarios should be fair
  - One requirement in correct solution
Machine Language Code

- What is atomic and what is not?
  - Assignment?  $X = Y$
  - Increment?  $X = X+1$

---

Algorithm 2.6: Assignment statement for a register machine

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$: load $R_1,n$</td>
<td>$q_1$: load $R_1,n$</td>
</tr>
<tr>
<td>$p_2$: add $R_1,#1$</td>
<td>$q_2$: add $R_1,#1$</td>
</tr>
<tr>
<td>$p_3$: store $R_1,n$</td>
<td>$q_3$: store $R_1,n$</td>
</tr>
</tbody>
</table>
Critical Reference

- Reference to variable \( v \) is critical reference, if …
  - Assigned value in \( P \) and read in \( Q \)
    - Read directly or in a statement
- Program satisfies limited-critical-reference (LCR)
  - Each statement has at most one critical reference
  - Easier to analyze than without this property
  - Each program is easy to transform into similar program with LCR

\[
\begin{align*}
\text{Not LCR:} & \quad n = n + 1; \quad n = n + 1 \\
\text{Not LCR:} & \quad n = m + 1; \quad m = n + 1 \\
\text{LCR:} & \quad \text{tempP} = n + 1; \quad \text{tempQ} = n + 1; \quad n = \text{tempP}; \quad n = \text{tempQ}; \\
\end{align*}
\]

LCR vs. atomicity? (ouch)
Volatile and non-atomic variables

- **Volatile variable**
  - Can be modified by many processes (must be in shared memory)
  - Advice for compiler (pragma)
    - Keep something in memory, not in register
    - Pseudocode – does not generate code

- **Non-atomic variables**
  - Multiword data structures: long ints, arrays, records, …
  - Force access to be indivisible (atomic) in given order

---

What if compiler/hw decides to keep value of \( n \) in a register/cache? When is it stored back to memory? What if \( \text{local1} \) & \( \text{local2} \) were volatile?

---

**Algorithm 2.8: Volatile variables**

\[
\begin{array}{c|c|c}
\text{Algorithm} & \text{Volatile variables} \\
\hline
\text{integer } \mathbf{n} & \leftarrow 0 \\
\hline
\text{p} & \text{q} \\
\hline
\text{integer } \text{local1, local2} & \text{integer } \text{local} \\
p_1: \mathbf{n} \leftarrow \text{some expression} & q_1: \text{local} \leftarrow \mathbf{n} + 6 \\
p_2: \text{computation not using } \mathbf{n} & q_2: \\
p_3: \text{local1} \leftarrow (\mathbf{n} + 5) \times 7 & q_3: \\
p_4: \text{local2} \leftarrow \mathbf{n} + 5 & q_4: \\
p_5: \mathbf{n} \leftarrow \text{local1} \times \text{local2} & q_5: \\
\end{array}
\]
Example Program with Volatile Variables

- Can implement it in any concurrent programming language
  - (Extended) Pascal and (Extended) C
  - BACI (Ben-Ari Concurrency Interpreter)
    - Code automatically compiled (from Extended Pascal or C)
  - Ada
  - Java

\[
\text{Algorithm 2.9: Concurrent counting algorithm}
\begin{array}{|c|c|}
\hline
\text{integer } n & \leftarrow 0 \\
\hline
\hline
\text{p} & \text{q} \\
\hline
\text{integer temp} & \text{integer temp} \\
\text{p1: do 10 times} & \text{q1: do 10 times} \\
\text{p2: temp } \leftarrow n & \text{q2: temp } \leftarrow n \\
\text{p3: n } \leftarrow \text{temp } + 1 & \text{q3: n } \leftarrow \text{temp } + 1 \\
\hline
\end{array}
\]
**Concurrent Program in Pascal**

```pascal
program count;
var n: integer := 0;

procedure p;
var temp, i: integer;
begin
  for i := 1 to 10 do
    begin
      temp := n;
      n := temp + 1
    end
end;

procedure q;
var temp, i: integer;
begin
  for i := 1 to 10 do
    begin
      temp := n;
      n := temp + 1
    end
end;

begin { main program }
  cobegin p; q coend;
  writeln(‘The value of n is ’, n)
end.
```

**n** is volatile, because... it is assigned in one thread, and read in the other
What if compiler optimized and kept n in a register? 
Lets hope not! 
(in ExtPascal or C-- 
global (volatile) variables are seemingly kept in memory by default)
with Ada.Text_IO; use Ada.Text_IO;

procedure Count is
  N: Integer := 0;

  pragma Volatile(N);

  task type Count_Task;
  task body Count_Task is
    Temp: Integer;
    begin
      for I in 1..10 loop
        Temp := N;
        N := Temp + 1;
      end loop;
    end Count_Task;

    begin
      declare
        P, Q: Count_Task;
      begin
        null;
      end;
      Put_Line("The value of N is " & Integer'Image(N));
    end Count;
Concurrent Program in Java

```java
class Count extends Thread {
    static volatile int n = 0;

    public void run() {
        int temp;
        for (int i = 0; i < 10; i++) {
            temp = n;
            n = temp + 1;
        }
    }
}
```

```java
public static void main(String[] args) {
    Count p = new Count();
    Count q = new Count();
    p.start();
    q.start();

    try {
        p.join();
        q.join();
    }
    catch (InterruptedException e) {
    }
    System.out.println("The value of n is " + n);
}
```

> javac Adder8.java
> java Adder8

How many threads really in parallel? • how to control it?

Execute on 8-processor vera.cs.helsinki.fi?

http://www.cs.helsinki.fi/u/kerola/rio/Java/examples/Adder8b.java
Run Multi-threaded Java

- Why different result?
- What is correct result?

Execute on 8-processor vera.cs.helsinki.fi?


```
kerola@vera:~/public_html/rio/Java/examples$ javac Adder8.java
kerola@vera:~/public_html/rio/Java/examples$ java Adder8
    finally n = 80000 = 37358
kerola@vera:~/public_html/rio/Java/examples$ java Adder8
    finally n = 80000 = 34464
```
BACI

- Ben-Ari Concurrency Interpreter
  - Write concurrent programs with
    - C-- or Ben-Ari Concurrent Pascal (.cm and .pm suffixes)
    - Compile and run in BACI
  - GUI for Unix/Linux

- jBACI
  - Just like BACI
  - GUI for Windows

- Installation
  - load version 1.4.5 jBACI executable files and example programs, unzip, edit config.cfg to have correct paths to bin/bacc.exe and bin/bapas.exe translators, click run.bat

- Use in class, homeworks and in project

18.1.2011 Copyright Teemu Kerola 2011
BACI Overall Structure

C-- to PCODE Compiler

add.pco

void main() {
  cobegin { add10(); add10(); }
  ....
}

add.cm

C-- (Concurrent C)

bacc.exe

add.lst

... void main() {
  17  24  cobegin {add10(); add10();}
  ....
}

Executing PCODE ...

C n = 1 i = A n = 1 C2 i = 1 A
C n = 4 i = 2 C
B n = A n = 5 i = 24 A

bainterpreter.exe


18.1.2011 Copyright Teemu Kerola 2011
jBACI

Just like BACI, but with Java

- requires Java v. 1.4 (SDK or JRE)
- Built-in compiler and interpreter
- edit state
- run state


18.1.2011 Copyright Teemu Kerola 2011
Add a breakpoint to selected (PCode) line
Summary

- Abstraction, atomicity
- Concurrent program, program state
- Pseudo-language algorithms
- High level language algorithms
- BACI