Concurrency at Programming Language Level

Ch 2 \[BenA 06\]

Abstraction
Pseudo-language
BACI
Ada, Java, etc.

Lesson 2

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
    - Can be at any granularity
    - Can trust on that atomicity

**Atomic Statement**

- Atomicity guaranteed somehow
  - Machine instruction: HW
  - Programming language statement, set of statements, or set of machine instructions
    - SW
      - Manually coded inside
      - Provided automatically to the user by programming environment

**Concurrent Program**

- Sequential process
  - Successive atomic statements
  - Control pointer (= program counter)

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

**Program State, Pseudo-language**

- **Sequential program**

  ```
  p1: n = k1
  p2: n = k2
  ```

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

- **(Global) Program State**

  ```
  p1: n = k1
  p2: n = k2
  q1: n = k1
  ```

  **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: value of each local/global/shared variable
- Nr of possible states can be (very) large
  - Not all states are reachable states
  - Different executions do not go through same states (even with same input)

State Diagram and Scenarios

- Transitions from one possible state to another
  - Contains all reachable states and transitions
  - All possible executions are included, they are all correct

Atomic Statements

- Two scenarios
  - Both correct
  - Different result!

Algorithm 2.1: Trivial concurrent program

Algorithm 2.3: Atomic assignment statements

- Scenario 1 — OK
  - 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?
- 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)
  - Invariant:
    - \( P \) will get his turn to eat
    - \( n \) value will become 2

- Liveness property
  - Property must eventually become true ("good" eventually happens)
  - Eventually the mouse cursor is not displayed

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
- Common Temporal Logic (CTL)
  - Can be checked automatically
  - Every time computation reaches given state
  - SMV model checker
  - NuSMV model checker

Machine Language Code

- What is atomic and what is not?
  - Assignment?
  - Increment?

\[
\begin{align*}
X &= Y; \\
X &= X + 1;
\end{align*}
\]

Algorithm 2.6: Assignment statement for a register machine

<table>
<thead>
<tr>
<th>( P )</th>
<th>( Q )</th>
<th>( a_i )</th>
<th>( b_i )</th>
<th>( c_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>load R1, n</td>
<td>load R1, n</td>
<td>load R1, n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R1, #1</td>
<td>add R1, #1</td>
<td>add R1, #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>store R1, r</td>
<td>store R1, r</td>
<td>store R1, r</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Critical Reference

- Reference to variable \( v \) is critical reference, if …
  - Assigned value in \( P \) and read in \( Q \)
  - Reference has at most one critical reference
  - Easier to analyze than without this property
  - Each program is easy to transform into similar program with LCR
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**
  - Multitword data structures: long ints, arrays, records, ...
  - Force access to be indivisible (atomic) in given order

---

Example Program with Volatile Variables

Algorithm 2.6. Volatile variables

```plaintext
int local1, local2;
int i = some expression;
store n?
end
```

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

---

Example Program in Pascal

```
program count;
{
procedures
procedure p;
var temp: integer;
begin
for i := 1 to 10 do
begin
store n?
end;
end;
begin
main program
end;
}
end.
```

```
possibly volatile
```

Concurrent Program in Ada

```
with Ada.Text_IO; use Ada.Text_IO;
procedure p is
begin
for i := 1 to 10 loop
begin
store n?
end loop;
end;
```

```
Concurrent Program in Java

```java
public static void main(String[]) args { 
    // static volatile int n = ...
    int temp;
    for (int i = 0; i < 10; i++) { 
        temp = i;
        n = temp + 1;
    }
    Thread.yield(); // force?
}
```

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

```
java Adder8.java
```

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

---

Copyright Teemu Kerola 2011

What if compiler optimized and kept n in a register? Lets hope not!

In ExtPascal or C the global (volatile) variables are seemingly kept in memory by default.

Concurrent Programming (RIO) 18.1.2011
Run Multi-threaded Java

```
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
```

Why different result?
What is correct result?

Run them yourself?
(Copy source code in your own directory)

BACI Overall Structure

```
add.cm
void main() {
    cobegin {add10(); add10();}
}
```

```
C- to PCODE Compiler
add.pco
```
Summary

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