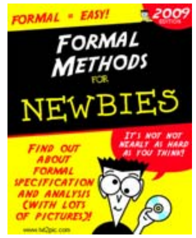
CISC422/853: Formal Methods in Software Engineering: Computer-Aided Verification



Topic 1: A few words about concurrency

Juergen Dingel
Jan, 2009

What is concurrency?

- **Concurrent programs…**

  consist of units (typically called threads, or processes) that
  - **on a multi-processor machine:** could be executed by different processors at the same time
  - **on a single-processor machine:** could be executed by different schedules in different interleavings
  - **communicate** through
    - shared memory or message passing
- Demos:

  garden1.java (2 threads, 1 shared object)
  [Kramer, McGee: “Concurrency: State Models and Java Programs” `http://www.doc.ic.ac.uk/~jnm/book/`]

  demo1.c (2 threads, no shared variables)

  demo2.c (2 threads, 1 shared variable)

What is concurrency? (Cont’d)

- **Concurrent programs**

  typically require
  - **synchronization** though, e.g.,
    - locks: e.g., one per object; only held by ≤ 1 processes
    - semaphores: natural number n together with two atomic operations:
      - P(n): if n>0, then n:=n-1; else suspend calling process
      - V(n): if some process p suspended on n, then resume p; else n:=n+1
    - monitors: abstract data type representing a shared resource
      - private monitor variables, monitor operations, condition variables
  - to prevent **interference** on shared data through **race conditions**
- Demo

  garden2.java (2 threads, 1 shared and synchronized object)

Why is it hard?

- Sequential programs special case of concurrent ones
  - Every concurrent program can be made to execute sequentially without much effort
    - **tradeoff:** amount of parallelism ⇔ risk of interference
    - **ideally:** program exhibits a maximal degree of concurrency, i.e., contains a minimal amount of synchronization
- Consequences:
  - Harder to write: When adding a line of code to
    - a **sequential** program, programmer must be aware of what
      - has happened up until that point, and
      - will happen after that point
    - a **concurrent** program, programmer must also be aware of what
      - may have or may not have happened concurrently

      ⇒ harder to get code to work correctly

Why is it hard? (Cont'd)

- Consequences (Cont'd)
  - Harder to debug:
    - When program doesn't work may be difficult to reproduce error
  - Harder to test:
    - Impossible to test program comprehensively with respect to all possible schedulings
  - Harder to reason about:
    - unexpected interference can lead to surprising results

> ***"I've come across many teams whose application worked fine even under heavy and extended stress testing, and ran perfectly at many customer sites, until the day that a customer actually had a real multiprocessor machine and then deeply mysterious races and corruptions started to manifest intermittently."***

[H. Sutter. "The free lunch is over: A fundamental turn to concurrency in software". Dr. Dobb's Journal, 30(3), March 2005]

Unexpected interference can lead to surprising results

- Consider the following concurrent program with shared variable x:

```
N := 5; x := 0;
[ for (i=0; i<N; i++) {        ||   for (j=0;j<N; j++) {   ]
      x := x+1;                          x := x+1;
  }                                  }
```

- What are the values that x can have upon termination?
  - When assignments are atomic: ?
  - When assignments are not atomic: ?
- What if N = 1000? N =10^8?
- Could you devise a comprehensive test that shows this?

Testing Doesn't Always Cut It

```
int x, y, tmp;

thread swap() {
   tmp = x;
   x = y;
   y = tmp;
}

proc p(int i, j) {
   // pre: i+j is odd
   ...
}

thread main() {
   x=1; y=2;
   run swap();
   run swap();
   p(x, y);
}
```

- Will the call p(x,y) always succeed?
- Can't exhaustively test for these kinds of race conditions, because
  - not enough control over relative execution speeds on multi-processor machines
  - not enough control over scheduling policy on single-processor machines
  - combinatorial explosion

But Wait, It Can Be Even Worse!

- **Ideally:** statements in a program are executed in the order in which they appear in the text
- **However:** this is disallows many performance-enhancing compiler optimizations (to, e.g., take advantage of parallelism, multi-level caches, optimistic execution)
- **Next best thing:** sequential consistency
  - *"every read of a variable will 'see' the most recent write in execution order to that variable by any processor"*
  - allows, e.g., "semantics-preserving" reordering
- Ok for sequential programming, but still too restrictive for concurrent programming

```
…
x = 1;
y = 2;
z = x+y;
…
```

```
…
x = 1;
y = 2;
z = x+y;
// x,y not
// aliases
…
```

But Wait, It Can Be Even Worse! (Cont'd)

- **Wanted:** Balance desires of program developer (ease of program development) with desires of compiler writer (optimization possibilities)
- **Enter:** the Java Memory Model (JMM)
  - Specifies minimal guarantees that JVM must make about when the effect of an action A of thread T1 are visible to action B of thread T2
  - A in T1 visible to B in T2 iff A in T1 *"happens before"* B in T2

Definition: A in T1 *happens before* B in T2 if
- T1 == T2 and A comes before B
- A == T.start() and B is an action in T
- …
- (A in T1 happens before C in T3) and (C in T3 happens before B in T2)

But Wait, It Can Be Even Worse! (Cont'd)

- **In summary:** JMM allows some surprising manipulations

```
public class PossibleReordering {
   static int x = 0,  y = 0;
   static int a = 0,  b = 0;
   public static void main(String[] args) throws InterruptedException {
      Thread one = new Thread(new Runnable() {
         public void run() {
            a = 1; x = b;
         }
      };
      Thread two = new Thread(new Runnable() {
         public void run() {
            b = 1; y = a;
         }
      };
      one.start(); two.start();
      one.join(); other.join(); System.out.println("( " + x + "," + y + ")");
   }
}
```

- What will this program output?
- Is this the only output it can produce?
- How do you test for this?

Demo: Reorder.java

But Wait, It Can Be Even Worse! (Cont'd)

- **In summary:**
  - JMM allows some surprising manipulations
  - Synchronization statements allow programmer to
    - restrict the "happens before" partial order, and thus to prevent certain, unwanted compiler manipulations to occur
- **By the way:**
  - reading up on, experimenting with, and summarizing the JMM would make a nice project

Why use concurrency?

- **Performance gain:**
  - ideally, speed up factor equal to number of processors
- **Ease of programming:**
  - concurrency is a real-world phenomenon
  - many programming problems are inherently concurrent and can be solved more naturally using concurrency:
    - E.g., embedded systems, reactive systems

And we have seen last time, an embedded system may be controlling the brakes in your car tomorrow…

Model checking

- Model checking helps us out here:
  - Exhaustive enumeration of all possible executions/schedules of a concurrent program
  - Check that all of them are ok

  $\Rightarrow$ complete confidence (when exploration was exhaustive)