Testability, controllability, observability

- Testability, controllability, observability
- A look at controllability and observability issues in:
  - real-time systems
  - multithreaded software

Eduardo Marques, edrdo@di.fc.ul.pt
Testability

The likelihood of exposing a fault through tests.

“assuming that a given software artifact contains a fault, how likely is it that testing will reveal that fault” [12 - A. & O., 9.2]
Measuring testability?

**Model of Testability**

*If a program has a fault, how difficult will it be to find a test that causes a failure?*

**Simple model**

\[
\text{Testability} = \frac{|\text{failure causing}|}{|\text{Input}|} \times 100 \%
\]

*Impractical to measure*
Approximating Testability

Testability can be approximated with the RIP model and mutation

- **R = % inputs from some distribution that reach X**
- **I = % inputs that cause a fault to infect (average over N faults)**
- **P = % infected states that propagate to output**

Sensitivity (X) = R * I * P

Testability (P) = F (Sensitivity (X)), for all X in P

Issues in Approximating Testability

- Reasonable input distribution?
- How to induce faults?
  - What faults?
- How to measure propagation?
  - Expensive!
- Information hiding reduces propagation
- Assertion checking can be used to increase testability
Controllability & observability

○ **Controllability:**
  ✪ Ability to affect the software behavior (in particular, replicate that behavior).
  ✪ “How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors.” [D1.12 - A. & O., p. 14]

○ **Observability:**
  ✪ Ability to observe software behavior
  ✪ “How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment, and other hardware and software components.” [D1.11 - A. & O., p. 14]

○ Low observability and/or controllability imply low testability.
How we dealt with observability/controllability … a few examples

- OO software and Test doubles, mock/spy objects in particular:
  - Mock objects let us control dependencies and stub behavior.
  - The use of mock or spy objects allow us to observe object interactions (e.g., using frameworks like Mockito)
  - A number of other issues arises - measuring testability/testing OO software can be extremely hard due to inheritance and polymorphism …

- Web applications
  - Frameworks like JWebUnit allow us to control/simulate browser interaction and observe/assert contents/results of that interaction.
  - Web applications usually incorporate multiple layers of functionality that challenge controllability and observability.

- Database testing
  - We’ve applied test patterns (e.g. backdoor manipulation ) and tools to control/populate the initial database contents and later observe/verify its contents.
Testability & real-time software

- Real-time software: software subject to timing constraints.
  - Example: Metronome real-time garbage collector from IBM: [http://www.youtube.com/watch?v=jid4WaILPBk](http://www.youtube.com/watch?v=jid4WaILPBk)

- **Hard real-time systems:**
  - Failure to meet a time constraint = fatal system failure (e.g., cardiac pacemaker)

- **Soft real-time systems**
  - Failure to meet a deadline is tolerable up to a certain level of performance degradation (e.g., video streaming, VOIP)
Real-time software is often part of an embedded system, that includes sensors and actuators to interface with physical environment.

Timing constraints are imposed by:

- the expected functionality, i.e., the “time contract” for the software in interaction with a physical environment
- ... but also due to constraints of the computational platform.

Scarceness of resources in computational platform, e.g., CPU speed, available memory, ...) impacts on the ability to meet the expected functionality.
Periodic real-time tasks

- A classic formulation — \textbf{program} = \{ periodic real-time tasks \}
- Each task $T = (P, D, E)$ is defined by a period $P$, a relative deadline $D$, and a worst-case execution time (WCET) $E$ ... [in the figure above $P=D$]
- Feasibility/scheduling problem: is there a schedule for the real-time tasks such that they all meet their deadlines?
  - Analysis can be done statically, as long as WCET upper bounds can be estimated with high confidence (something that is quite complex)
Testability & real-time software

- **Observability issues**
  - Testing typically involves instrumenting a real-time program with "probes" to measure timing behavior.
  - ... but the insertion of "probes" may in turn alter the timing behavior.

- **Controllability issues**
  - The execution time of a task is sensitive to many low-level details such as the memory hierarchy state (e.g. cache state), context switches, I/O ...
  - Simulation of sensor readings/physical actuation in embedded software.
Testability & multithreaded software

- Multithreaded programs
  - multiple threads of control
  - shared memory between threads
  - use of multithreading primitives: locks, barriers, ...

- Failures in multi-threaded software due to thread interaction:
  - races
  - deadlocks
  - livelocks
  - ...

Testability & multithreaded software

- **Controllability issues**
  - Preemptive scheduler performs context switches in non-deterministic, irreproducible manner
  - How to test a particular schedule of interest?
  - What schedules are actually of interest?

- **Observability issues**
  - Debugging a program affects its schedule ...
  - Failures are hard to replicate ...
  - "Heisenbugs": “a bug that disappears or alters its behavior when one attempts to probe or isolate it”

- Many bugs are elusive to detect and reproduce.
  - e.g., see “Concurrent Bug Patterns and How to Test Them”, E. Farchi, Y. Nir, S. Ur, PDPS, 2003
Dining philosophers problem

- **N** philosophers sit at a round table to think and eat spaghetti. One fork is placed in between each philosopher.
- To eat a philosopher must grab the fork to his left, then the fork to its right.
- Deadlock results if all philosophers pick their left fork at the same time.
Philosopher thread

class Philosopher implements Runnable {
    Object left, right; /* forks */
    boolean thoughts = false;
    boolean food = false;
    Philosopher(Object lf, Object rf) {
        left = lf;
        right = rf;
    }
    public void run() {
        thoughts = true; // 1. think
        synchronized (left) { // 2. get left fork
            synchronized (right) { // 3. get right
                food = true; // fork and eat
            } // 4. release right fork
        } // 5. release left fork
    }
    boolean hadThoughts() { return thoughts; }
    boolean hadFood() { return food; }
}
Dining philosophers’ test

@Test
public void testDinner() {
    Object forks[] = new Object[N];
    for (int i = 0; i < N; i++)
        forks[i] = new Object();
    Philosopher[] philosophers = new Philosopher[N];
    for (int i=0; i < N; i++)
        philosophers[i] = new Philosopher(forks[i], forks[(i+1) % N]);
    runThreads(philosophers);
    for (int i=0; i < N; i++) {
        assertTrue(philosophers[i].hadThoughts());
        assertTrue(philosophers[i].hadFood());
    }
}

How is a deadlock reached?

Suppose we can monitor the threads and check for failure due to deadlock ... will the test always fail? Will the test ever fail?
It turns out that very rarely the test will fail (deadlock is reached)...

- \(\sim 1\) every 20000 executions for \(N = 2\) (2 threads) on Java 7, even less for \(N > 2\) [Marques et al., PPPJ’14]

- A very precise (& difficult to replicate) schedule is required for failure = low testability.
Example 2

```java
public class Semaphore {
    int count;
    Semaphore(int initial) { count = initial; }
    int getCount() { return count; }
    void down() throws InterruptedException {
        synchronized (this) {
            while (count == 0) {
                wait();
            }
            count--;
        }
    }
    void up() {
        synchronized (this) {
            count++;
        }
        if (count == 1) {
            synchronized (this) {
                notify();
            }
        }
    }
}
```

Unsynchronized access to `count` leads to data races and possible deadlock. Why?
Observation: context switches are relevant only at points that cause thread interference, e.g.,
- lock acquisition and release (Java: enter/leave synchronized blocks)
- use of multithreading primitives (Java: wait(), notify(), ...)
- shared data access

Approaches
- Randomised scheduling: introduce “noise” at thread interference points, e.g. using small delays, to increase likelihood of context switches. This is non-deterministic, but increases the likelihood of failure detection. [Example tools: rstest, ConTest ]
- Cooperative scheduling: make threads voluntarily yield at interference points; code between yield points executes serially; the scheme is deterministic and can be used to enumerate all possible schedules [Example tools: CHESS, Cloud9, CONCURRIT, Cooperari].