



# Coverage Tidbits

Shmuel Ur  
HVC 2008



## Coverage: introduction

- ◇ What is coverage?
  - ◇ For a given system, define a set of testing tasks.
    - ◇ Code coverage uses the code to define the tasks.
    - ◇ Functional coverage uses the application description to define the tasks.
  - ◇ For each task, check if it was actually performed (covered) in some test.
- ◇ **Coverage is a Good Thing.** It is the main tool for –
  - ◇ Finding areas that need additional testing.
  - ◇ Evaluating the quality of testing.
- ◇ Obtaining certain coverage criteria is required by many standards and company policies.
- ◇ Many coverage tools exist, for all popular programming languages and environments.



## Coverage models hit parade

- ◇ Some coverage models are very popular:
  - ◇ statement
  - ◇ branch-point
- ◇ Others are less popular:
  - ◇ multi-condition
- ◇ Yet others are rarely used:
  - ◇ define-use
  - ◇ mutation
  - ◇ path
- ◇ Why?



## A coverage model will be widely accepted if...

1. Tasks are generated **statically**.
  - ◆ *Coverage percentage* can be measured.
2. Each task is **well-understood** by the user.
  - ◆ True for most models, but not all.
3. Almost all tasks are **coverable**; for the few tasks that are not, the programmer is able to tell why they are not coverable.
  - ◆ True for statement coverage, but not for define-use.
  - ◆ Otherwise, the tester spends too much time trying around and investigating.
4. Each uncovered task yields an **action item**.
  - ◆ Uncovered statement → either redundant code, or insufficient test.



## Coverage for concurrent testing

- ❖ Concurrent programs are notoriously bug-prone.
- ❖ Concurrent testing is notoriously hard.
- ❖ 100% statement coverage is far from guaranteeing thorough testing.
- ❖ Good concurrent coverage is called for.



## Synchronization coverage – definition

- ❖ Test that each synchronization primitive in the code did “**interesting**” stuff.
- ❖ synchronized blocks in Java:

```
synchronized(lock1) {  
    counter++;  
    updateDB();  
}
```

```
synchronized(lock1) {  
    counter--;  
    updateDB();  
}
```

- ❖ Two tasks for each synchronized block:
  - ❖ **blocked**: A thread waited there, because another thread held the lock.
  - ❖ **blocking**: A thread holding the lock there caused another thread to wait.



## Java synchronized blocks tasks

↓  
`synchronized(lock1) {`  
    ↓  
    `counter++;`  
    `updateDB();`  
}

- ☒ blocking
- ☒ blocked

↓  
`synchronized(lock1) {`  
    ↓  
    `counter--;`  
    `updateDB();`  
}

- ☒ blocking
- ☒ blocked



## Meets the requirements:

- ✓ Tasks are generated statically from the code.
- ✓ Each task must be well-understood by the developer/tester:
  - ◆ Less trivial than for statement coverage, but certainly not too difficult for a reasonable concurrent programmer.
- ✓ Each task must be coverable:
  - ◆ A synchronized block is written in order to make threads wait. If it can't happen, perhaps it's redundant.
  - ★ Sometimes it's not easy to make a concurrent task happen, but this is **precisely the purpose**: make an effort to make the concurrent test thorough.





## Getting organization to adopt coverage

- ◇ Must be easy to use
  - ◇ Need to work very hard on being easy to use ☹
- ◇ Needs to be meaningful to the people using it
  - ◇ For example, system test can not use statement coverage
- ◇ The result should be visible to everyone
  - ◇ The “shame” factor
- ◇ The impact on performance/memory should be tolerable
- ◇ Once measured there are many side benefit