Application of Synchronization Coverage

Arkady Bron, Eitan Farchi, Yonit Magid,
Yarden Nir-Buchbinder , Shmuel Ur

PPoPP 2005

Presented in the spring 2011 Seminar on Advanced Topics in Concurrent Programming (236802) by

Orna Agmon Ben-Yehuda

Outline

- Testing and Coverage
- Synchronization coverage
- Use Cases
- Back to the Future

Testing

- Testing usually takes 40%-80% of the development time.
- Bug detection rate drops as the code improves.
- How good are the tests? Can the client quantify the quality of the tests?
- When should testing cease?
  - Testing for regression suits must be brief and to the point
- Where should testers focus efforts?

Coverage

- What is coverage?
  - For a given system, define a set of testing tasks.
  - For each task, check if it was actually performed (covered) in some test(s).
  - Measure the fraction of tasks performed.
- Coverage is the main means for
  - Quantifying testing quality (contracts, company policies)
  - Targeting future test creation
  - Guiding code review

Sequential Coverage Models Hit Parade

- Some coverage models are very popular:
  - Statement – each code line is a task. DA requires 100% statement coverage.
  - Branch-point – each decision outcome (e.g., the two branches of “if”) is a task
- Others are less popular:
  - Multi-condition – combinations of decision outcomes form single task
- Yet others are rarely used:
  - Define-use – each pair of data definition and use is a task
  - Mutation – the extent to which tests can discriminate the code from slight mutations. Errors are seeded in the code. Discovering each error is a task. The fraction of discovered seeded errors is an indication for the number of real bugs still left in the code.
  - Path – a possible flow of the code from entry point to exit point is a task.
- Why?

A coverage model will be widely accepted if...

- Tasks are **statically generated** from the code.
  - *Coverage percentage* can be measured.
- Each task is **well-understood** by the user.
  - True for most models, but not all.
- Almost all tasks are **coverable**;
  - for the few tasks that are not, the programmer can tell why.
  - True for statement coverage, but not for define-use, mutation, multi-condition.
  - Otherwise, the tester wastes time investigating.
- Each uncovered task yields an **action item** (Either redundant code or missing tests).

Coverage for Concurrency

- Concurrent programs are bug-prone and concurrent testing is hard.
- A combination of sequential and concurrent coverage is required to verify testing adequacy (Factor et al.).
- 100% statement coverage is far from guaranteeing thorough testing:
  - Good programmers try to make synchronized code parts small.
  - On a single processor, it means a low probability for a thread to lose the processor while still in the synchronized code.
  - In 2005, threads were often used on a single core.
  - Removing all synchronization commands from a concurrent code, and testing until full statement coverage was reached often resulted in hardly discovering any bugs.

Coverage for Concurrency

Good concurrent coverage is called for

No concurrent cov. model meets requirements

No concurrent cov. Model is widely used

1. Concurrent Pair of Events

- A task is a pair of code lines which were consecutively executed in a test, with an additional field:
    - “False” if they were executed by the same thread
    - “True” otherwise
- Too many tasks – full coverage is hard
- Which tasks are coverable?
- Used for evaluating testing progress

2. Shared Variables

- A task is at least two threads touching instances of the same variable.

- Which tasks are coverable?

Synchronization Coverage!

- A task is a synchronization primitive in the code which does something "**interesting**".
- 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.
- To cover the code:
  - Create a list of synchronization code areas
  - List tasks for each code area
  - Full coverage is when all tasks were tested for all sync. code areas

Example: Java synchronized blocks tasks

```
synchronized(lock1) {
```
synchronized(lock1) {

```
    counter++;
    updateDB();
}
```

- [x] blocking
- [x] blocked

```
synchronized(lock1) {
```
synchronized(lock1) {

```
    counter--;
    updateDB();
}
```

- [x] blocking
- [x] blocked

Other Tasks: Try Lock

- Task “Failed”
- Task “Succeeded”
- If a task never happens: Insufficient testing or a redundant sync operation

Other Tasks: Wait (on a condition)

- Task "Repeated" (wait was called at least twice in the same block, in the same run, by the same thread)
- Wait proper use is in a loop
- Bug pattern: calling wait and not verifying the condition still holds upon wake-up.
  - Wake-up may happen spuriously in some systems
  - Another thread may have invalidated the condition between notification and wake-up

Other Tasks - Future Work:

- Semaphore – Wait:
  - Task “Blocked” (semaphore was not immediately available), task “Non-Blocked”
- Semaphore –Try Wait:
  - Task “Succeeded” , task “Failed”
- Notify (NotifyAll, Signal, signalAll, Broadcast):
  - Task “Had Target”, task “Had no Target”
  - “Lost Notify” bug pattern: wait called after notify, notify had no target, program hangs
  - Testing for “had no target” may be hard, but usually possible. Requires deep understanding.

Other Tasks – Bad Ideas:

- Task “interrupted” for Wait:
  - Usually uncoverable

Synchronization coverage 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 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.

and...

- ✓ Each uncovered task yields an action item:
  - If the synchronization is redundant, remove it.
  - Otherwise, some interleaving scenario has not been tested. Strengthen the test.

- ☹ Few tasks genuinely cannot be covered.
  - Peculiarities of the synchronization protocol.

Uncoverable synchronization tasks

```
public static void main (String args) {
    ...
    synchronized (lock) {
        new Thread(...).start();
        ...
    }
}
```

- If thread..start() waits for lock, it will always be blocked, main will always be blocking.
- This is rare; such tasks can individually be identified when reviewing the uncovered tasks.
- (Same for statement coverage.)

Implementation: ConTest

- IBM tool: ConTest (concurrent testing tool).
- Implemented for Java and for C/Pthread.
  - The code is instrumented, and a list of the tasks is computed.
  - In test runtime, it keeps a representation of the synchronization objects, e.g., in order to know when a lock is held.
  - **Noise injection mechanisms** (yields and sleeps) help obtaining the tasks.
- The concept can be adopted for any set of synchronization primitives (Windows, Java 5.0 new concurrency library).
  - Task definition for each primitive depends on primitive semantics.

Field pilot #1: Unit-test of a real-life concurrent protocol

- VOD system.
  - Heavy stream operations by some threads
  - Business logic by other threads, with real time interleaving
  - Business logic is an inner class of the Thread Manager class, written from scratch
  - 160 classes
- A synchronization protocol (Thread Manager) was isolated and abstracted.
  - Tester class implemented
  - Heavy functions replaced by random choices or constant results (must be done with protocol understanding!)
  - Several bugs fixed using various other methods.
- Tested with ConTest.
  - 100% sync. coverage reached after less than 1 hour.
  - This gives confidence in the quality of the protocol.
  - Without noise injection, most of the tasks would not have been performed.
  - No additional bugs found.

Field pilot #2: synchronization coverage to guide review

- A system to identify complex events in distributed context, and give warnings.
  - For example, warn that many resources are near their limit.
- Synchronization coverage – initially 25%.
  - 575 classes , 16 with sync primitives (not too many, but a lot to review)
  - Can the number of sync files be less?
  - 9 classes with 3 or less sync primitives
  - 4 1-sync-primitive-files: 2 different code base, 1 sync not needed
- One-hour review of uncovered tasks quickly revealed –
  - 2 tasks in dead code.
  - 2 unnecessary synchronizations.
  - 4 tasks : missing tests.
  - 1 task hard-to-test – carefully review.
  - 1 task: missing the application code (missed due to abstraction)
  - Few bugs
- Measurement #2: 39%.

What happened

Since 2005?

ConTest

- ConTest is an active product and service of IBM. A 10-class version is available for free, a larger version is a commercial product.
- http://www.alphaworks.ibm.com/tech/contest
- ConTest now checks if locks are taken in a reverse order, thus possibly leading to a deadlock (source: conversation with Shmuel Ur).
- Chess

Citations (June 2011)

- According to ACM: 18 citations
- 6 self
- 5 by Yuanyuan (YY) Zhou
- For example:

SyncFinder (OSDI 2010)

- Ad hoc syncronization considered harmful (incurs about 50% chance of bugs, hard to review)
- SyncFinder finds ad-hoc sync points, thus enabling tools like ConTest to work on them.
- But do not use them!

CTrigger (ASPLOS 2009)

- Exposing atomicity bugs in real software (Apache, MySQL, Mozilla)
- 2-4 orders of magnitude faster than stress testing.
- Fast bug reproducibility without the limitation of a single thread at a time.
- Focuses on unrealizable interleavings, deliberately inducing the rare interleavings.

Kim, Cho, Moon 2011

- Re-implemented Sync. Coverage using a deterministic technique.
- 1.5-144 times faster than a random approach, with no abstraction.
- Only checked simple codes (fft, lu-decompose, basic compression, etc.)

Conclusion

- A good concurrent coverage model, and a supporting tool, were strongly needed.
- Synchronization coverage met this need.
- In 2005, the authors had reasons, both theoretical and practical, to believe that it can be widely used in the industry.
- In 2011 we can see that the authors prediction came to pass, on the basis of publications and available tools.

Bibliography

- http://www.testingstandards.co.uk/living_glossary.htm
- M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka. *Testing concurrent programs: a formal evaluation of coverage criteria*. In Proceedings of the Seventh Israeli Conference on Computer Systems and Software Engineering, 1996.
- Weiwei Xiong , Soyeon Park , Jiaqi Zhang , Yuanyuan Zhou , Zhiqiang Ma, *Ad hoc synchronization considered harmful,* Proceedings of the 9th USENIX conference on Operating systems design and implementation, 2010.
- Soyeon Park, Shan Lu, Yuanyuan Zhou: *CTrigger: exposing atomicity violation bugs from their hiding places*. ASPLOS 2009.
- HyoYoung Kim, DaeHyun Cho, Sungdo Moon, *Maximizing synchronization coverage via controlling thread schedule, IEEE CCNC 2011.*