Application of Synchronization Coverage

Michael Riabzev
Technion – Israel Institute of Technology
January 4, 2018

based on paper by Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur from IBM Haifa Research Labs
Denver International Airport
Background

- Designed in the late 80’s
- To be largest airport in USA (140Km$^2$)
- Challenge: transferring baggage quickly
- Solution: largest automated baggage handling system
  - using sophisticated underground tracks and carts system
Complexity

- Completely automatic system
- Carts collect the baggage directly from the plane
- Using barcode readers the cart is directed to required destination
- Synchronization is fine enough so the cart doesn’t even stop collecting baggage, arriving just in time as it falls out of the plane

size:
  - $200M
  - 700,000 Km of cables
  - 40 Km of underground tracks
  - 300 controlling computers
Figure: Denver international airport illustration (Dr. Dobb’s Journal)
Chaos

- Something went wrong on demonstration day (March 94')
Chaos

- Something went wrong on demonstration day (March 94’)
  - EVERYTHING
Chaos

- Something went wrong on demonstration day (March 94’)
  - EVERYTHING
- Speeding carts fell of tracks
- Luggage was thrown on tracks without arriving cart
- Luggage was crashed and ruined by hitting carts
- Carts crashed into each other
- Barcodes where miss-read, sending luggage to wrong terminal
Cost

- Opening delayed by 16 months, costing $1.1M
Cost

• Opening delayed by 16 months, costing $1.1M per day
• Most of the automatic system was removed, urgently replaced by infrastructure for the common manual solution ($311M)
• Only United-Airlines used the automatic system (until 2005)
  ○ With maintenance cost of $1M per month

Conclusions:
- You better test your program
- Synchronization is hard, you better test it twice
Cost

- Opening delayed by 16 months, costing $1.1M per day
- Most of the automatic system was removed, urgently replaced by infrastructure for the common manual solution ($311M)
- Only United-Airlines used the automatic system (until 2005)
  - With maintenance cost of $1M per month
- Conclusions:
  - You better test your program
  - Synchronization is hard, you better test it twice
Program testing
Program Correctness challenges

- General code analysis is impossible
  - \( HP \not\in \text{RE} \)
- Provable correctness of “common” programs
  - achievable using formal-verification
  - high overheads, often infeasible
Program testing

• correctness promised only for a few inputs...
  ◦ ... often negligible fraction
• quality depends on many factors
  ◦ e.g. chosen tests, assumptions in code and tests, coverage
• based on heuristics
  ◦ no "correct way", only "good ideas"
• Testing is \( \sim 40 \text{ – } 80\% \) of development process
Test coverage analysis

- A way to measure how thorough tests are
- Verify each statement is executed by some test
- If this is not the case:
  - There is potential untested possible flow, or
  - there are unused blocks of code
- Win-Win
Coverage for Concurrency

- Concurrent code is hard $\implies$ fragile implementation
- Concurrent testing is hard $\implies$ buggy delivered system
- Statement coverage is not perfect, worse with concurrency
  - Why?
Synchronization Coverage

• Idea: check if synchronization primitives fulfill their purpose
  ○ e.g. some thread was actually blocked by *lock* instruction

• If this is not the case the reasons might be:
  ○ Redundant synchronization instruction $\implies$ causing inefficiency
  ○ Potentially dangerous concurrent scenario is not tested

• Win-Win
Implementation

- The paper was published at 2005
- It reported internal usage of the system (in IBM)
- Currently there is a public system based on it (ConTest)
Unit testing with synchronization coverage - VOD

- High throughput rendering pipeline
  - \( \sim \) 20 frames per second
- A mature project of \( \sim \) 160 classes
- A few bugs were found using synchronization coverage test
  - after less than one hour!
Conclusions
• Verifying code correctness is hard
• Verifying correctness of concurrent code is very hard
• Testing synchronization coverage is a good idea:
  ○ Does not require much resources
  ○ Might prevent embarrassment
  ○ Helps one focus on fragile functionality, requiring thorough review
• Verifying code correctness is hard
• Verifying correctness of concurrent code is very hard
• Testing synchronization coverage is a good idea:
  ○ Does not require much resources
  ○ Might prevent embarrassment
  ○ Helps one focus on fragile functionality, requiring thorough review

Thank you!