Reachability testing for concurrent programs

Yu Lei and Richard Carver
Presented by Thuan Huynh
Overview

• Introduction
• Some existing tools
• Reachability testing
  – Concepts
  – Algorithm
  – Implementation
  – Optimizations
  – Results
• Conclusion
Concurrent programs

• Multiple non-independent executions
  – Multithreaded programs
  – Distributed programs
• Very difficult to test
  – Non deterministic interleavings/irreproducible

```
Thread1              Thread2              Thread3
  t.send(1)           t.send(2)            x = t.recv()
y = t.recv()
print x - y
```

– Difficult to breakdown because problems come from interactions
Approaches to testing

• Deterministic testing
  – Run all possible interleavings (how?)
  – Select a subset of interleavings and force execution to follow

• Non-deterministic testing
  – Run repeatedly for some time
  – Easy but inefficient, problems may appear at only extreme conditions at customers’ computers

• Prefix-based testing
  – Run test deterministically at the beginning
  – Follow by nondeterministic runs
Model checking/SPIN

• Use a modeling language PROMELA
• Explore all possible states of a program
• Support full LTL logic
• Suffer state explosion problem
  – Partial order reduction to relieve the problem
  – Use for very critical portion of software
  – Verify network protocols
Java PathFinder

- Formal verification tool developed by NASA Ames Research center
- A more easier to use SPIN
- Explore ALL possible execution paths of a java program without recompling
  - Also visit all possible states of the program
  - Check every state for violations of assertions/properties/exceptions/deadlocks/livelock
  - Has a lot of heuristics and optimization to work with big programs.
- **VeriSoft** for C/C++
Concutest-junit

• A concurrency-aware version of junit developed at Rice University

• Improvements:
  – Catch errors in auxiliary threads
  – Have new invariants to check threading related problems
  – Can insert delays at critical places
  – Can record and playback specific interleavings
ConTest

• A tool to test concurrent java programs developed by IBM Haifa Research Lab

• Works without recompiling/new test
  – Instruments existing bytecode
  – Inserts heuristic sleep() and yield() instructions to expose problems
  – Run multiple times
Reachability testing
(prefix-based testing)

• Concepts
• Algorithm
• Implementations
• Optimizations
• Results
SYN-sequence

- We only care about the order of operations whose interleavings has effect on execution
  - Sending/receiving data with another thread
  - Semaphore/Monitors
- General execution model: send/receive
- SYN-sequence: sequence of synchronization events
- Aim: execute all possible SYN-sequences
Happen-before relation

- Gives us the order of events, usually partial.
- We can extract these relations by watching an execution.
- The unordered events are subjected to testing.
- Why vector clock but not single global clock?
Partial order reduction
Algorithm (RichTest)

• Run and collect a SYN-sequence $s^*$
• $S \leftarrow \{s^*\}$
• Repeat
  – Get a sequence $s \leftarrow S$
  – Runs each variant of $s$ to collect sequences $s_1, s_2, \ldots, s_m$
  – $S \leftarrow \{s_1, s_2, \ldots, s_m\}$
Until $S = \text{empty}$
Example

Thread 1
P2.send(a)

Thread 2
x=p2.recv();
y=p2.recv();
p3.send(c);

Thread 3
u=p3.recv()
v=p3.recv()
p3.send(d);

Thread 4
p2.send(b)
p3.send(d);
More concepts

• Race condition: A receive() operation may match with different send()’s

• Race_set(r): all send events that can possibly be matched with the receive operation r
Race table

Contains one column for each receive event $r$ that has a nonempty $\text{race}_\text{set}(r)$. The numbers in each row represent
- $-1$: remove $r$
- $0$: no change
- $1..|\text{race}_\text{set}(r)|$: match $r$ to the $i^{\text{th}}$ send in $\text{race}_\text{set}(r)$
Example

Thread 1
P2.send(a)

Thread 2
x=p2.recv();
y=p2.recv();
p3.send(c);

Thread 3
u=p3.recv()
v=p3.recv()
p3.send(d);

Thread 4
p2.send(b)
p3.send(d);

\[
\text{race_set}(r1) = \{s1,s2\} \\
\text{race_set}(r3) = \{s3,s4\}
\]
Implementation

• Library of synchronization objects:
  semaphores, monitors, send, receive
• Control/record the execution using the library
• No modification to thread scheduler
  – Portable to other operating systems and languages
Optimization

- Aim: Do not visit a SYN-sequence twice
- Keeping a list of visited SYN-sequence is expensive
- Trick: only include variants that obeys a specific set of rules. Proven that
  - We can still visit all SYN-sequences
  - Can start from any SYN-sequence
  - Computationally inexpensive to check
Results
Results

![Graph 1: Transitions vs Philosophers]

- POR
- RT

![Graph 2: Executions vs Philosophers]

- POR
- RT
Conclusion

• The new method for reachability testing
  – Guarantees the execution of every SYN-sequence exactly once
  – Does not require keeping a list of all visited SYN-sequences
  – Outperforms existing partial order reduction based techniques
  – Is platform independent