VeriTrace: Towards Automatic Testing and Verification of Concurrent Programs

ZHANG Yu

LCS, ISCAS

Suzhou • November 2012

Motivation

- Concurrent programs are notoriously hard to write correctly.
- Testing and debugging of concurrent programs are also hard.
  – Bugs may not recur due to non-determinism.
  – Enumerating all possible executions is hard or practically impossible.
- There are not many practical tools for testing and verifying concurrent programs.
- We also need a tool for teaching purpose at CAS.

Concurrent Objects

- Our tool runs on JVM.
- We talk about objective concurrent programs
  - Concurrent objects are shared by multiple processes or threads.
  - Threads can call methods to read information and make change to the shared object.
  - Method execution takes time, and execution time in different threads can overlap.
- What does it mean for concurrent objects and methods to be correct?

ISCAS

Correctness for Concurrent Programs

- Sequential consistency
  - Every concurrent execution has a consistent sequential execution.
  - The sequential execution preserves the program/ execution) order in every single process.
- Linearizability
  - Stricter than sequential consistency
  - The sequential execution preserves the happen-before order between all process.
- Quiescent consistency
- Looser consistency at hardware/architecture level.

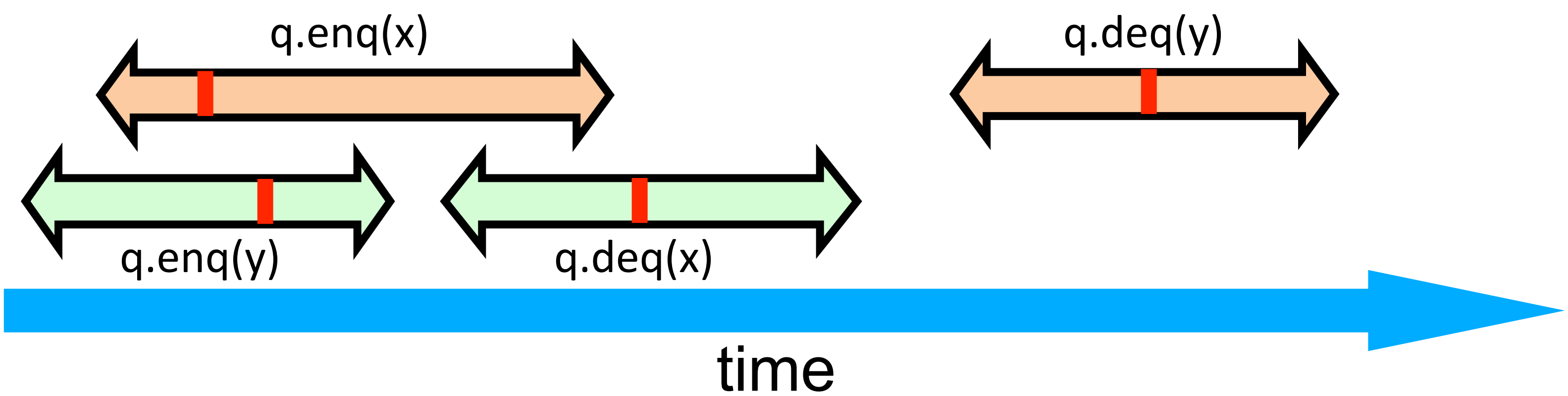
Linearizability

- Linearizable execution
  - Every method should “take effect” instantaneously between invocation and response.
  - Concurrent execution is correct if this sequential execution is correct.

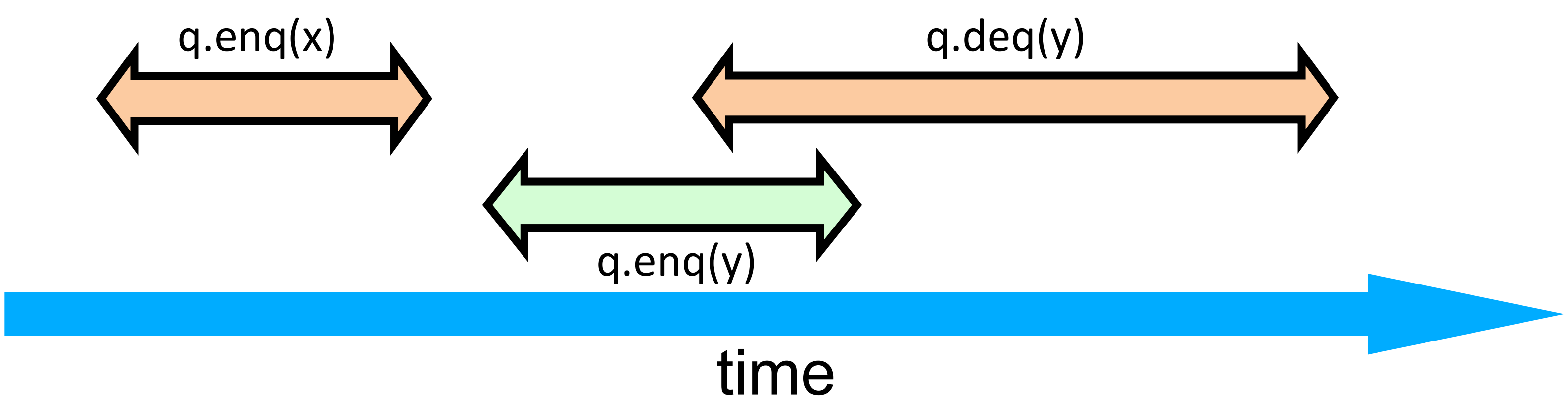
- A concurrent object is linearizable if all possible executions are linearizable.

Linearizability

- Linearizable execution:



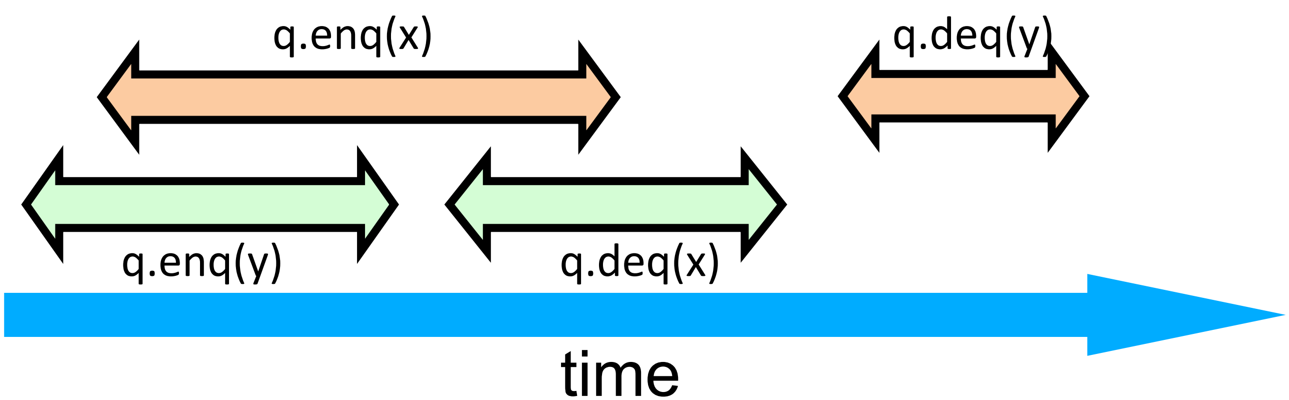
- Non-linearizable execution:



Trace Model

- We record traces of method executions in all threads
  - Every method execution has two events
    - Method call/invocation: method name & arguments
    - Method return/response: result or exception
  - A concurrent trace records method events tagged by thread ID, in temporal order.
  - Happen-before: a method execution m_1 happens before m_2 if m_1's response is before m_2's invocation in the trace.

B:q.enq(y)
A:q.enq(x)
B:q.enq:void
B:q.deq()
A:q.enq:void
B:q.deq:x
A:q.deq()
A:q.deq:y



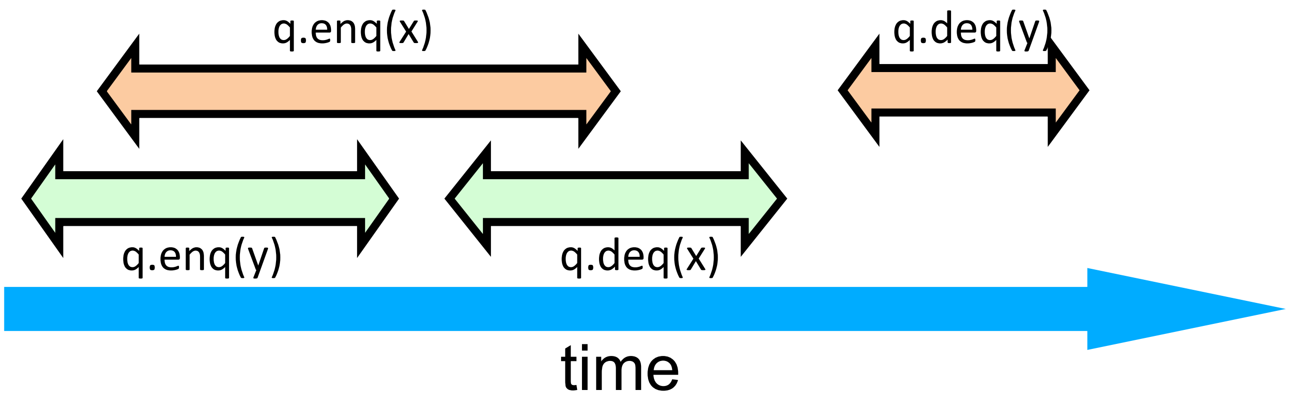
Serialization

- A simulation trace is a sequence of method calls and arguments.
  - E.g., {q.enq(x), q.enq(y), q.deq(), q.deq()}
  - Simulation traces are intended to be executed in a single-process mode.

- Given a concurrent trace, serialization produces all its possible simulation traces, preserving happen-before relation between method executions.
  - Executing a simulation trace produces a sequential trace, which is a sequence of method calls and results.
  - A sequential trace matches its original concurrent trace if every method execution returns the same result as in the concurrent one.

Serialization

B:q.enq(y)
A:q.enq(x)
B:q.enq:void
B:q.deq()
A:q.enq:void
B:q.deq:x
A:q.deq()
A:q.deq:y



- Its sequential traces are

A:q.enq(x):void	B:q.enq(y):void	B:q.enq(y):void
B:q.enq(y):void	A:q.enq(x):void	B:q.deq(): y
B:q.deq(): x	B:q.deq(): y	A:q.enq(x):void
A:q.deq(): y	A:q.deq(): x	A:q.deq(): x

  – Only the first trace is a matching trace.

Correctness

- Correctness in our model:
  - A concurrent trace is correct if its serialization has a matching sequential trace.
  - A concurrent program is correct if all its concurrent traces are correct.

Concurrent Trace Generation

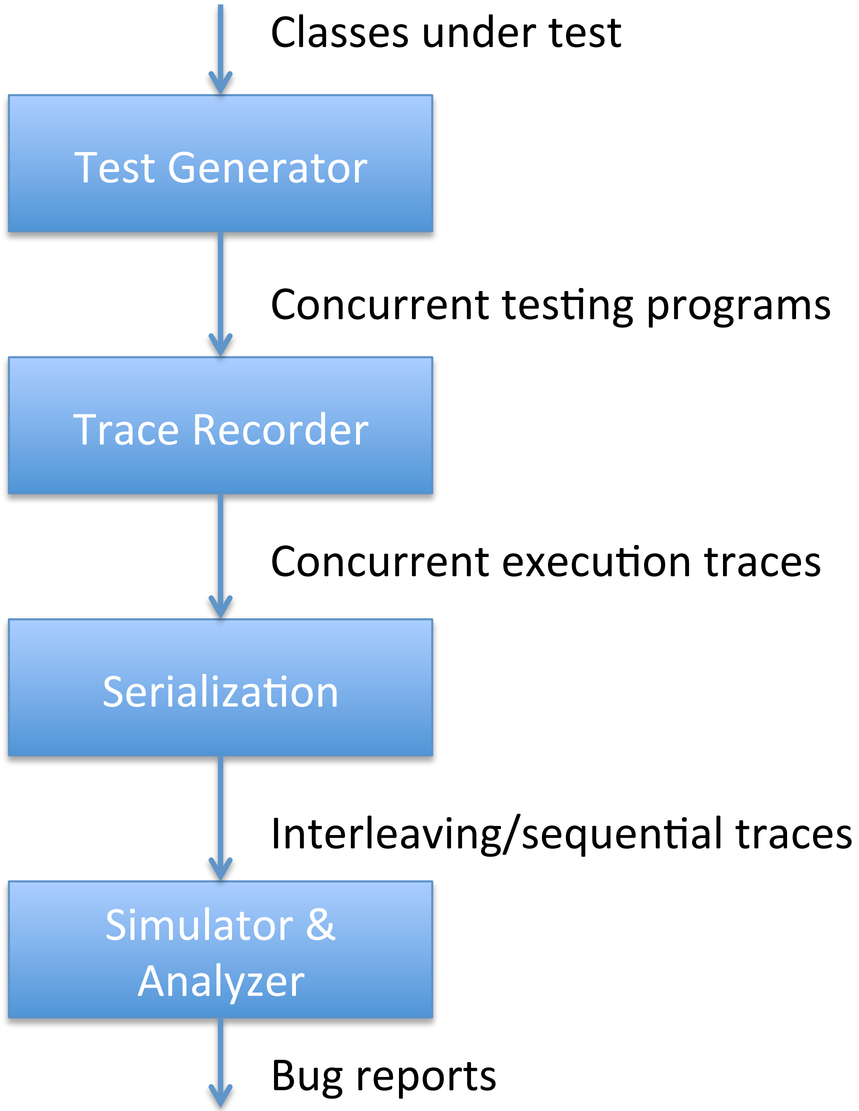
- Enumerating all possible concurrent traces is hard or practically impossible.
- We generate traces via testing: given a concurrent class for verification, a test case includes
  - a number of threads, and
  - a random sequence of method calls (and arguments) for each thread.
- We can record concurrent traces for every test case at the JVM level.
  - No annotation is required for source programs.
  - Test and trace generation are fully automatic.

Simulation and Analysis

- With each concurrent trace,
  - Serialization: produce all its simulation traces.
  - Simulate: execute the methods in the simulation trace in a single-thread manner, and check the result against the concurrent trace.
  - If no matching sequential trace is found, report the test with the buggy trace.

- The report is true negative: we do not produce false alarm.

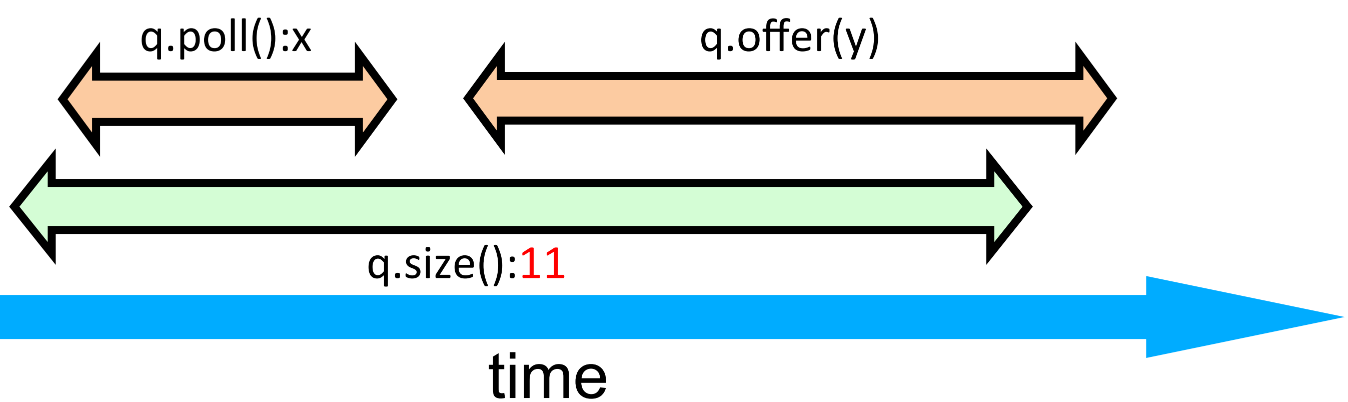
Design of VeriTrace



Experiments

- java.util.concurrent.ConcurrentLinkedQueue (2 threads)
  - Methods: offer/poll/size, 1000 tests
    - 200 method calls per thread, 2~3 buggy traces
    - 500 method calls per thread, 10~20 buggy traces
  - All buggy traces has the pattern: size || poll; offer
  - Method size is NOT thread-safe: it is stated in JSE7, but not in JSE6.

Assume size is 10 before execution

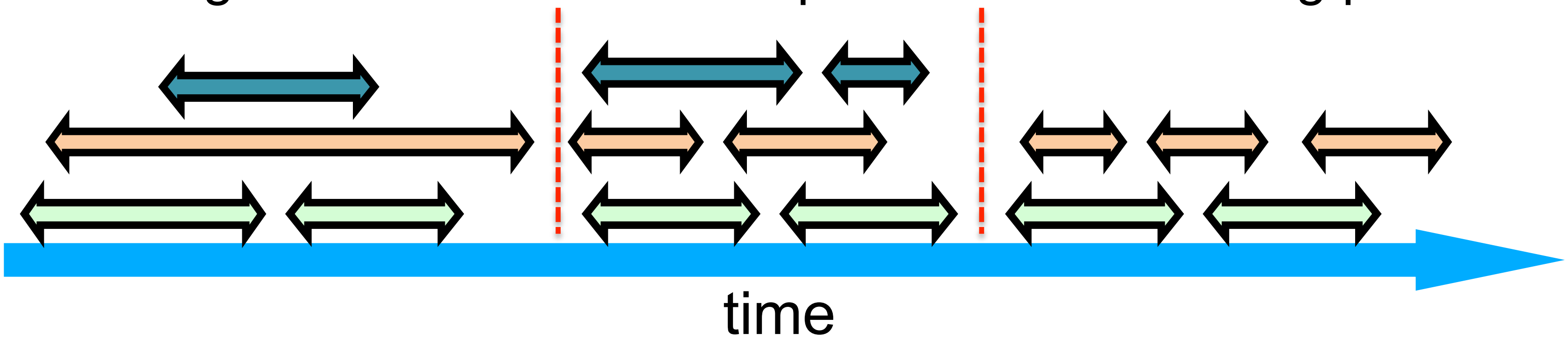


Experiments

- LockFreeList (Chapter 9, "*The Art of Multiprocessor Programming*")
  - Methods: add/remove, 1000 tests
    - 200 method calls per thread, ≤ 10 buggy traces
    - 500 method calls per thread, 30 ~ 40 buggy traces
  - All buggy traces has the pattern: remove || remove
  - Method remove is buggy, which is indicated by the online errata of the book.
  - The report also shows that data race only occurs between removes.

Implementation Issues

- Shallow simulation vs. deep simulation
  - A long concurrent trace often presents the following pattern:



  - Deep simulation produces full traces and starts simulation from scratch.
    - With a lazy interleaving algorithm it does not require much memory.
  - Shallow simulation do simulation for each segment and records intermediate states.
    - It often finds bugs quickly, faster than deep simulation.

Ongoing and Future Work

- The tool is still under intensive development.
- On-going work:
  - Use the tool to verify practical concurrent libraries.
  - Extend testing module to allow user-defined test cases.
  - Compare with other tools, e.g., LineUp, Thread-Safe.
- Future work:
  - Further analysis on buggy traces. Can help identify buggy methods?
  - Refined trace recoding to provide more information with buggy traces.
  - Combined with other program analysis techniques, e.g., model-checking, static analysis, etc.