Testing monadic programs using QuickCheck and state machine based models

Stevan Andjelkovic

2018.2.23, BOBKonf (Berlin)

Introduction

- Property based testing in general
- How to apply property based testing to stateful and concurrent programs
- Running example, simple CRUD web application
  - Familiar type of program
  - Obviously stateful and concurrent
  - Non-obvious property?
- Using the `quickcheck-state-machine` Haskell library, but the principles are general

Overview

- Reminder
  - What are property based tests?
  - Why are they so effective?
- Basic idea and motivation behind how the library applies property based testing principles to stateful and concurrent programs
- Demo
  - Sequential property (catches logic and specification bugs)
  - Concurrent property (catches race conditions)
- Comparison to other tools

Property based testing

- Unit tests

```
test :: Bool
test = reverse (reverse [1,2,3]) == [1,2,3]
```

- Property based tests

```
prop :: [Int] -> Bool
prop xs = reverse (reverse xs) == xs
```

Property based testing

- Unit tests

```
test :: Bool
test = reverse (reverse [1,2,3]) == [1,2,3]
```

- Property based tests

```
prop :: [Int] -> Bool
prop xs = reverse (reverse xs) == xs
```

- Proof by (structural) induction

  $\forall xs(\text{reverse}(\text{reverse}(xs)) = xs)$

Property based testing

- Unit tests

```
test :: Bool
test = reverse (reverse [1,2,3]) == [1,2,3]
```

- Property based tests

```
prop :: [Int] -> Bool
prop xs = reverse (reverse xs) == xs
```

- Proof by (structural) induction

  $\forall xs(\text{reverse}(\text{reverse}(xs)) = xs)$

- Type theory

```
proof : forall xs -> reverse (reverse xs) == xs
```

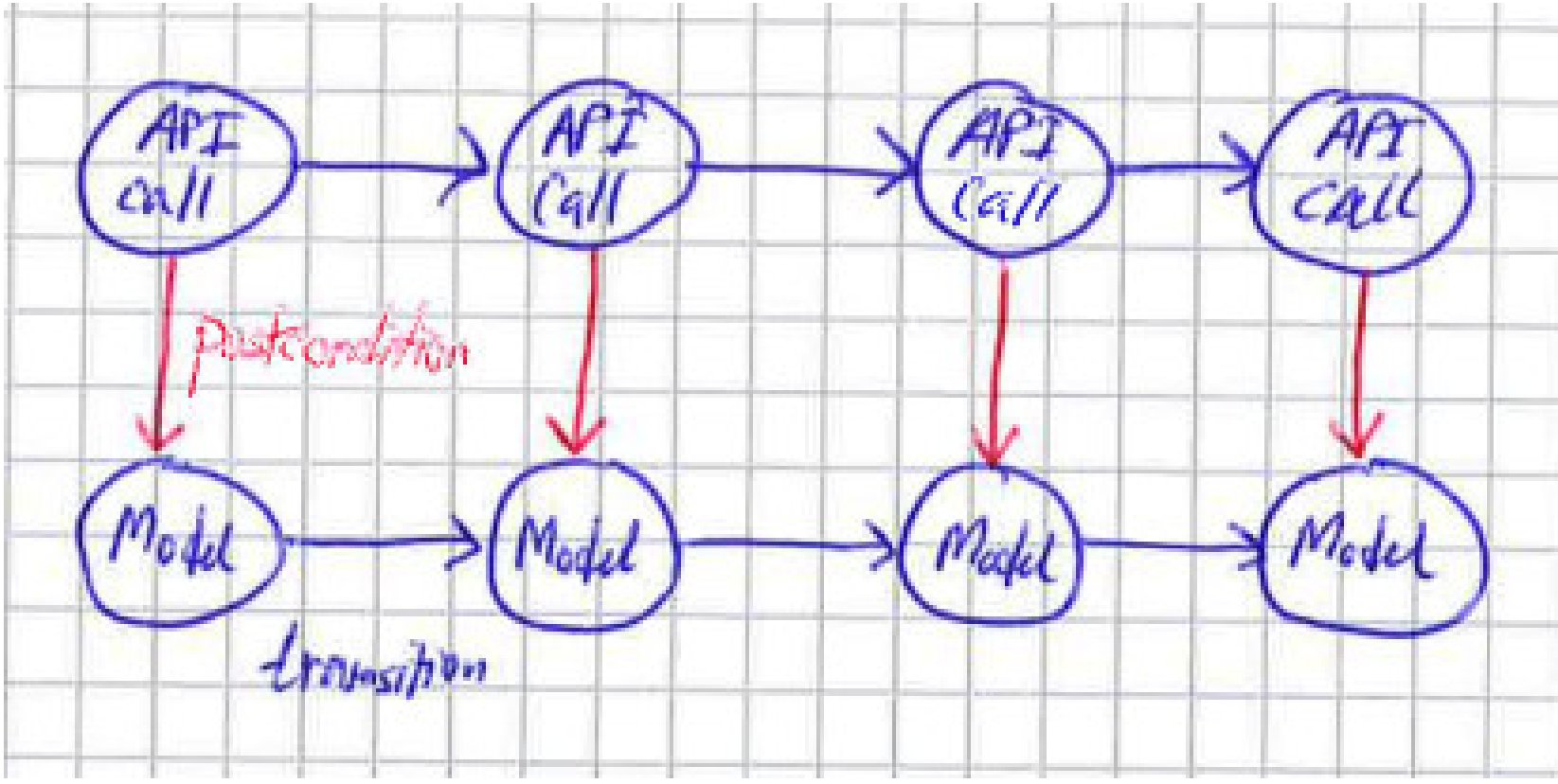
Stateful programs, basic idea / motivation

- Take inspiration from physics
  - Simplified model of reality that can predict what will happen
  - Experiments against reality validate the model
- How do we model algorithms/programs?
  - Gurevich's abstract state machines/new thesis, think of finite state machines where the states are arbitrary datatypes
- Abstract state machines are used by:
  - Quiviq's closed source version of QuickCheck for Erlang (Volvo cars, ...) (Claessen et al. 2009)
  - Z/B/Event-B familiy (Paris metro line 14)
  - TLA+ (AWS, XBox)
  - Jepsen (MongoDB, Cassandra, Zookeeper, ...)

The `quickcheck-state-machine` library

- Use abstract state machine to model the program
  - A model datatype, and an initial model
  - A datatype of actions (things that can be happen in the system we are modelling)
  - A transition function that given an action advances the model to the next state
- A semantics function that takes an action and runs it against the real system
- Use pre- and post-conditions on the model to make sure that the model agrees with reality (Floyd 1967; Hoare 1969)
- Use QuickCheck's generation to conduct experiments that validate the model
  - Sequential property
  - Parallel/concurrent property (linearisability, Herlihy and Wing 1990)

State machine model



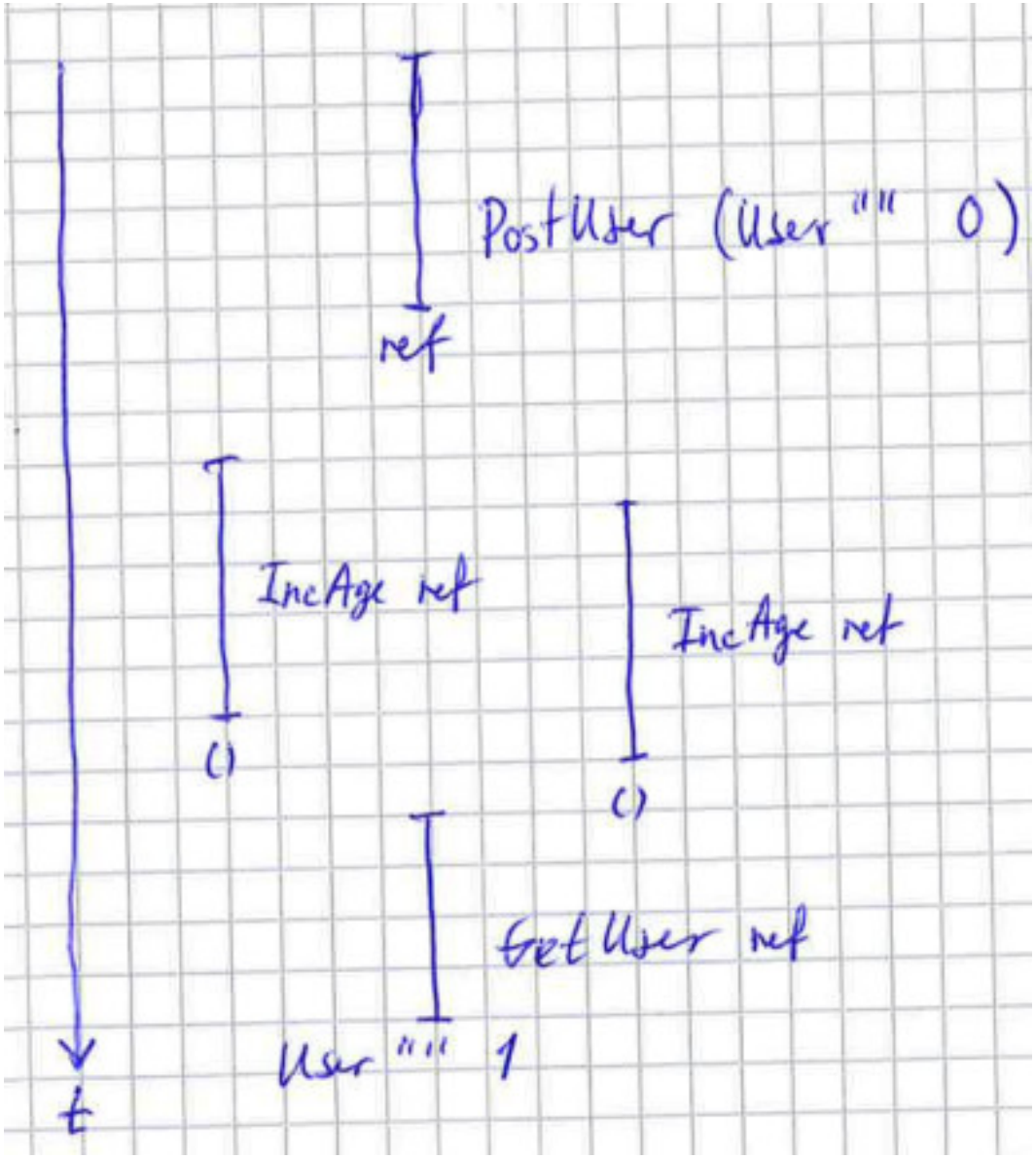
Demo

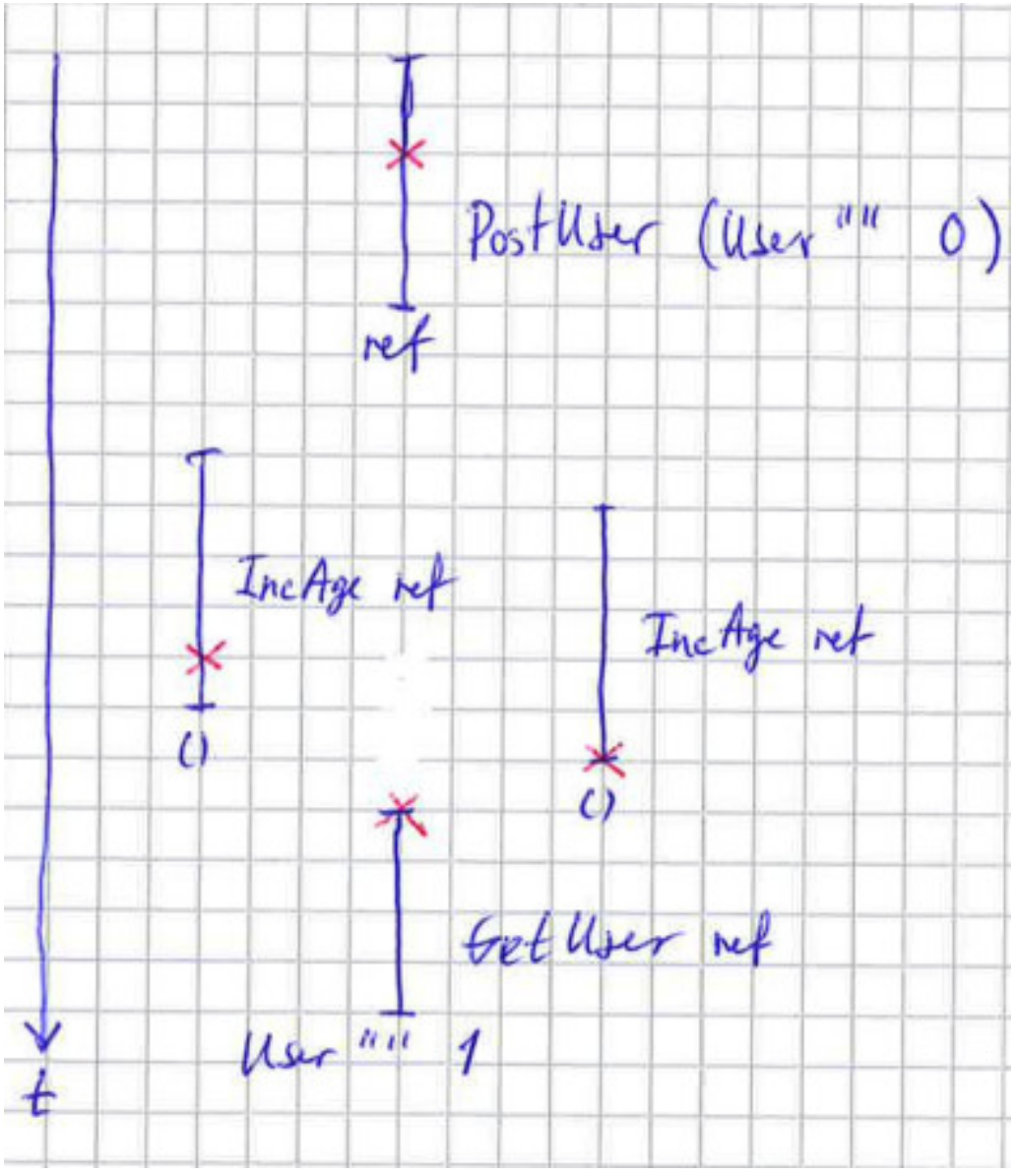
- Simple web application
  - `data User { name :: Text, age :: Int }`
  - Create user (post request)
  - Read/lookup user (get request)
  - Update age (put request)
  - Delete user (delete request)
- Implementation
  - Servant and persistant
  - Could be written using any libraries or language
  - Completely independent of `quickcheck-state-machine`
- Specification
  - Uses the `quickcheck-state-machine` library in the way described above

Demo

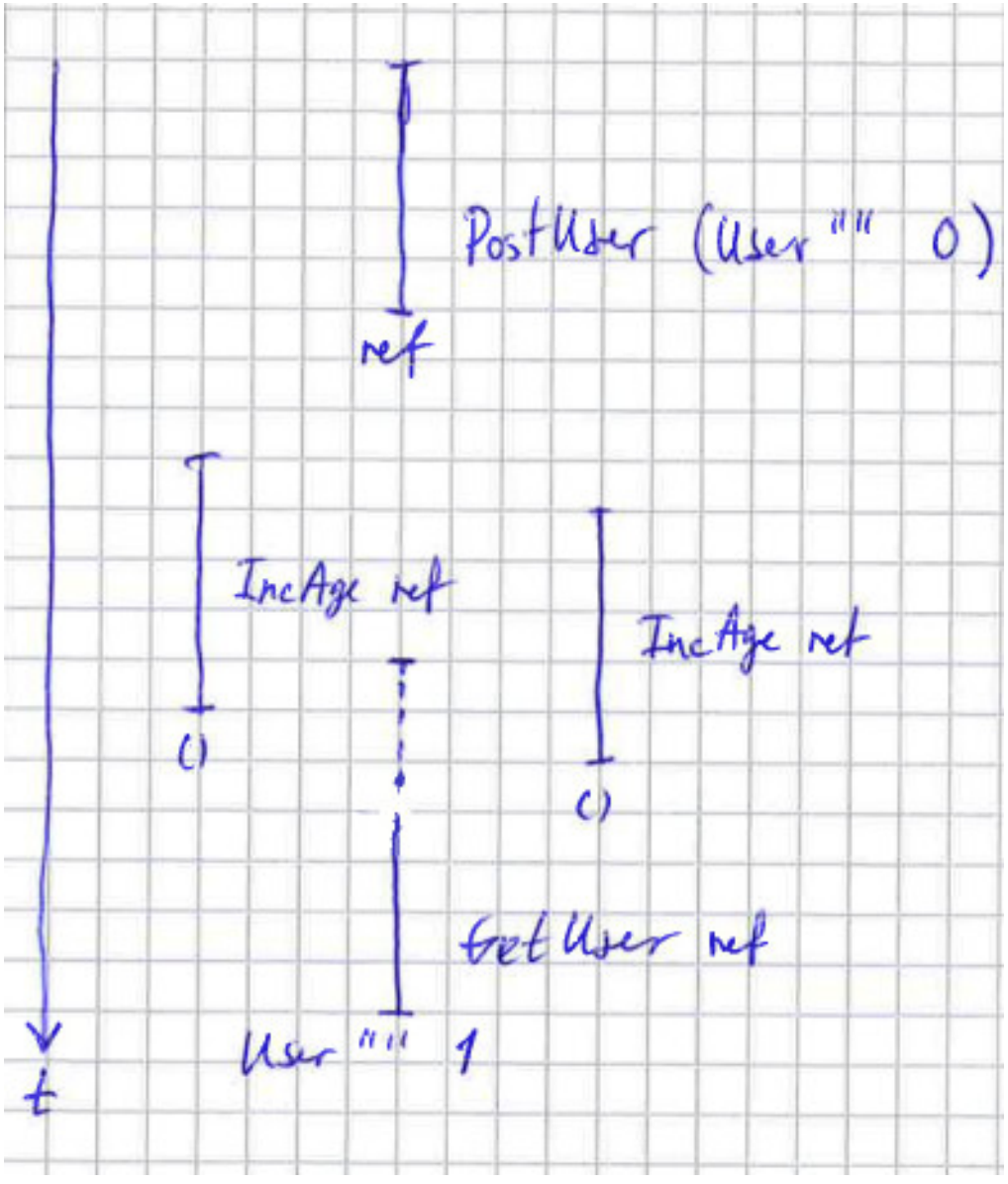
Linearisability (fails)

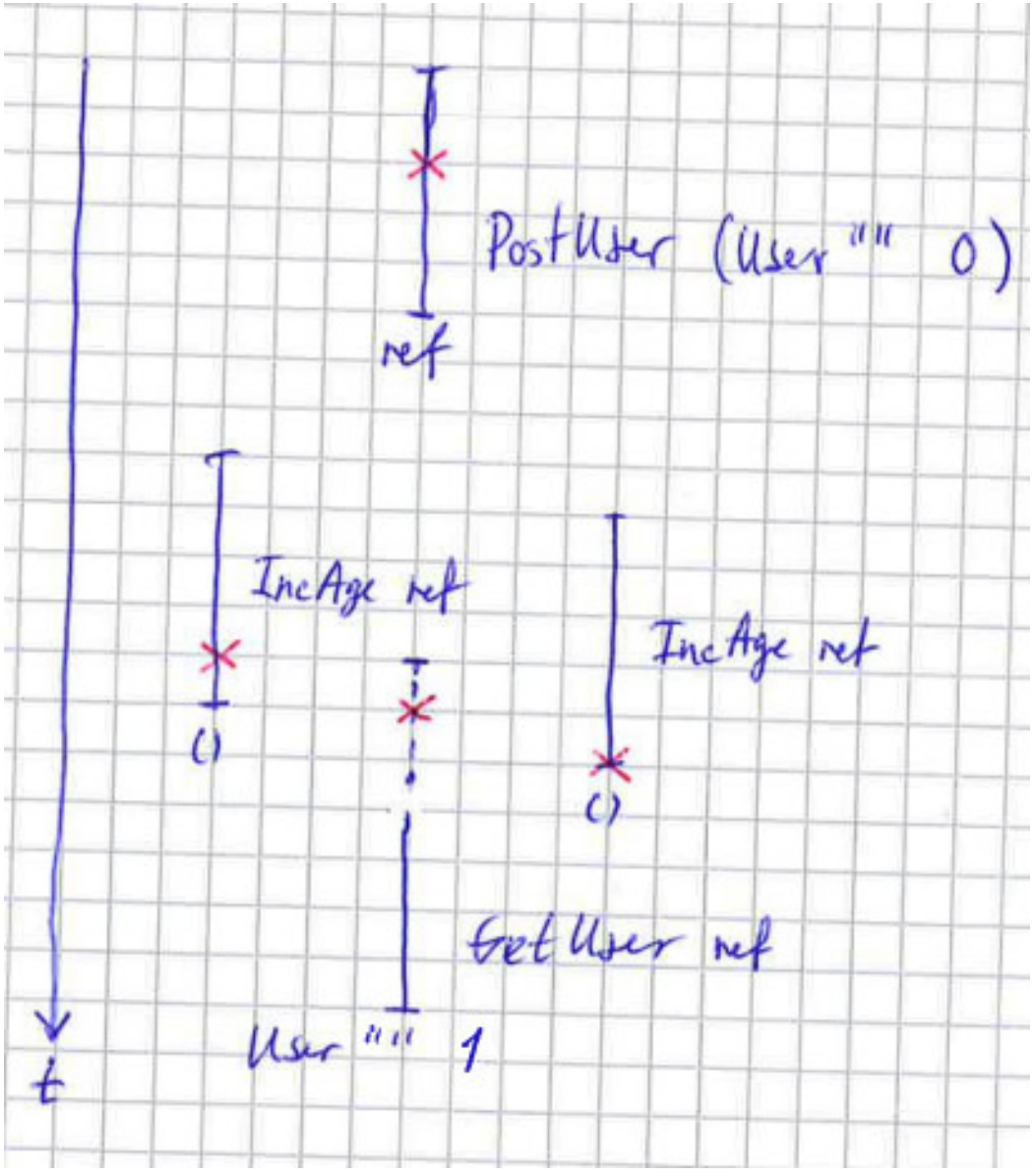
- (Herlihy and Wing 1990)




PostUser (User "" 0)
ref
IncAge ref
()
IncAge ref
()
GetUser ref
User "" 1
t

Linearisability (succeeds)




PostUser (User "" 0)
ref
IncAge ref
()
IncAge ref
()
GetUser ref
User "" 1
t

Comparison to other tools

- Quiviq's Erlang QuickCheck
  - More polished and used
  - Better statistics
  - Closed source
- Z/B/Event-B
  - Deductive proving (heavily automated)
  - Refinement
  - Notation
- TLA+
  - Model checking (proving is also possible)
  - Liveness and fairness properties can be expressed
  - No connection to actual implementation
- Jepsen
  - Does fault injection (e.g. network partitions)
  - No shrinking (is it even possible?)

Conclusion

- State machines are useful for modelling programs
- Race condition testing for free via linearisability
- See also
  - `quickcheck-state-machine` library on GitHub
  - Oskar's talk
  - Matthias' tutorial

References

Claessen, Koen, Michal H. Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf T. Wiger. 2009. "Finding Race Conditions in Erlang with QuickCheck and PULSE." In *Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, edited by Graham Hutton and Andrew P. Tolmach, 149–60. ACM. doi:10.1145/1596550.1596574.

Floyd, R. W. 1967. "Assigning Meanings to Programs." In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, edited by J. T. Schwartz, 19–32. Providence: American Mathematical Society.

Herlihy, Maurice, and Jeannette M. Wing. 1990. "Linearizability: A Correctness Condition for Concurrent Objects." *ACM Trans. Program. Lang. Syst.* 12 (3): 463–92. doi:10.1145/78969.78972.

Hoare, C. A. R. 1969. "An Axiomatic Basis for Computer Programming." *Communications of the ACM* 12 (10): 576–80, 583.