# Using Contracts to Guide the Search-Based Verification of Concurrent Programs

Chris Poskitt and Simon Poulding

ETH zürich  THE UNIVERSITY of York

SSBSE 2013, Saint Petersburg

25th August 2013

# Concurrency: necessary but challenging

- concurrency is necessary to exploit modern multicore architectures
- shared resources protected typically by an arbitration (locking) mechanism
- but this might introduce nasty non-functional faults
  - deadlock, livelock, starvation...
  - common symptom: lots of waiting

# How to show the absence of such faults?

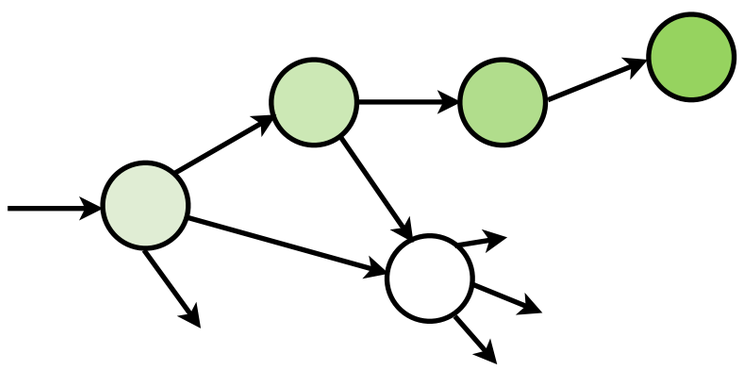
- must consider both input data (control flow) and temporal orderings of events (e.g. locking)
- exhaustive automatic techniques like model checking explore all possible executions

but real systems face a combinatorial blow-up of the state space

# STATE EXPLOSION

# More realistic: we can search for faults
## (without guaranteeing we find them all)

- metaheuristic search can be used to evolve strategies for navigating the state space
    - *Alba et al., Godefroid & Khurshid, Shousha et al., Staunton & Clark*
- find regions more "likely" to exhibit deadlock etc.
- but fitness functions usually coarse-grained
    - maximising number of blocked threads
    - minimising number of outgoing transitions

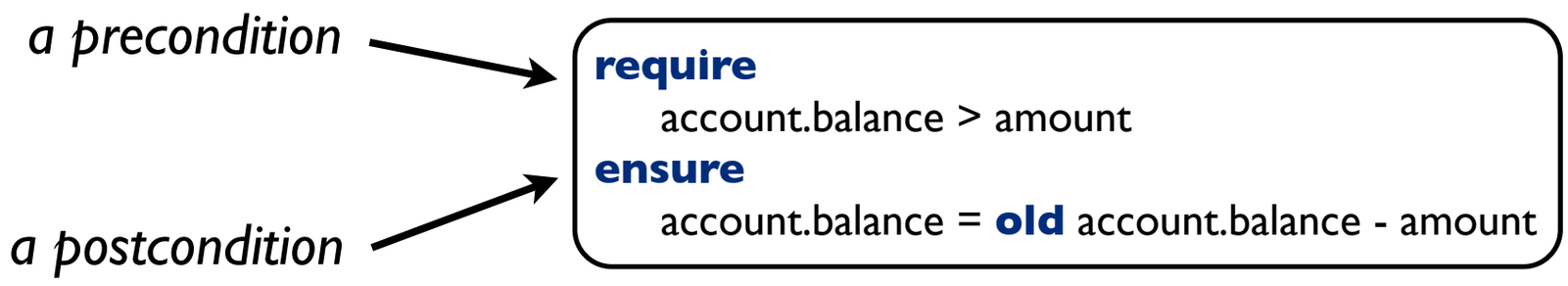


# There is an opportunity to provide more guidance

- utilise developer-specified contracts:
  executable pre- / postconditions and invariants

- not full mathematical specifications; we propose to use the Design by Contract style (Eiffel, JML, ... )

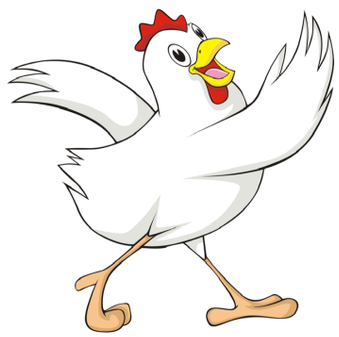
  e.g. a bank account withdrawal method



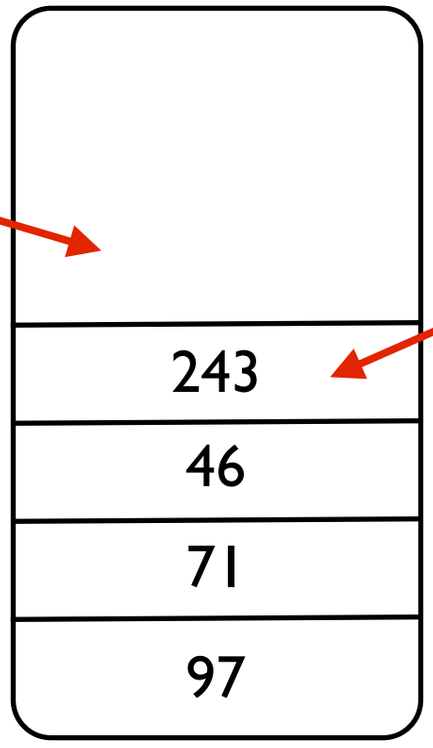

- contracts provide information that might not easily be inferred automatically

# Example: producers and consumers

(SCOOP and Java versions in the paper)

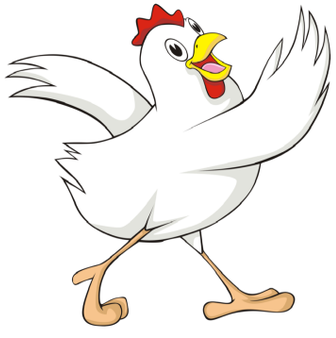


store (*buffer*, *int*)




consume (*buffer*)

*Producers*

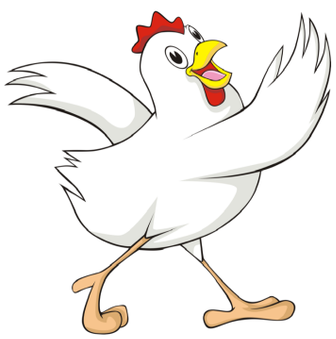


*Bounded Buffer*

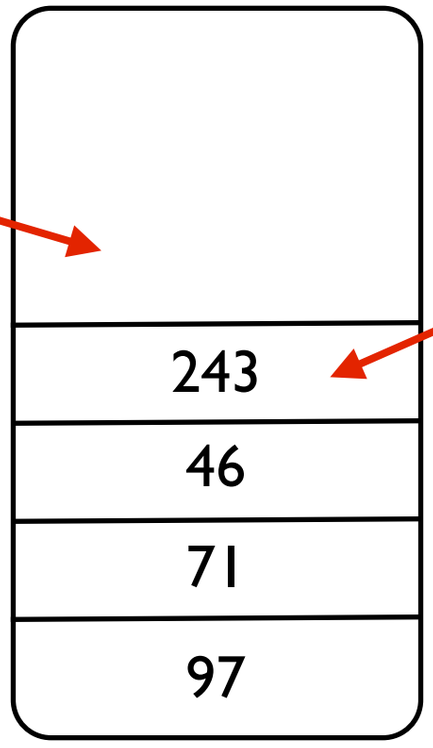
*Consumers*

# Example: producers and consumers

(SCOOP and Java versions in the paper)



store (*buffer*, *int*)




consume (*buffer*)

**require**
buffer.count < buffer.size

**require**
buffer.count > 0

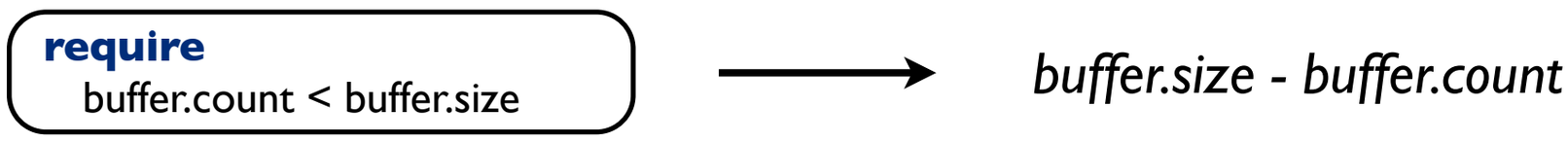
*Bounded Buffer*

# Example: producers and consumers

- the preconditions imply regions of interest in the state space

- program behaviours change over such boundaries
  - store might wait indefinitely if consumers are starved of access to full buffers
  - blocked producers may never be awoken if consumers fail to notify that the buffer is no longer full

- guide towards states that come closer to false valuations, using Korel/Tracey-like distance metrics



# In a nutshell 

We propose that additional information from (functional) contracts can be exploited to search for non-functional faults more effectively

# Ongoing work; next steps

- we are working on a prototype using Java PathFinder
- we are investigating how to exploit programs with lots of contracts
  - our example today considered two contracts in isolation
  - but how do we integrate *several* (possibly conflicting) contracts automatically and sensibly into fitness functions?
  - needs to be self-adaptive?
- what about other kinds of contracts?
  - temporal, resource usage, safety guarantees

# *That's all folks!*



CME: Concurrency Made Easy

DAASE: Dynamic Adaptive
Automated Software Engineering