Concurrency Requirements

Hubert Matthews, Oxyware Ltd
ACCU Conference, April 2006
hubert@oxyware.com

Why this talk?

- ACCU Conference 2004 - Andrei’s talk
  - 1960s: semaphores, mutexes, critical sections
  - 2004: semaphores, mutexes, critical sections
- Why no progress?
  - Is it just hard?
  - Do we lack the right languages and tools?
  - Are we thinking about it in the wrong way?
- Emphasis usually from implementors’ viewpoint
  - But what do users want and expect?

Reqts

merge conflict resolution

cache coherency

system of record

replication

stale data

versioning

batch update

synchronisation

problem space

Implementations

locks

mutexes

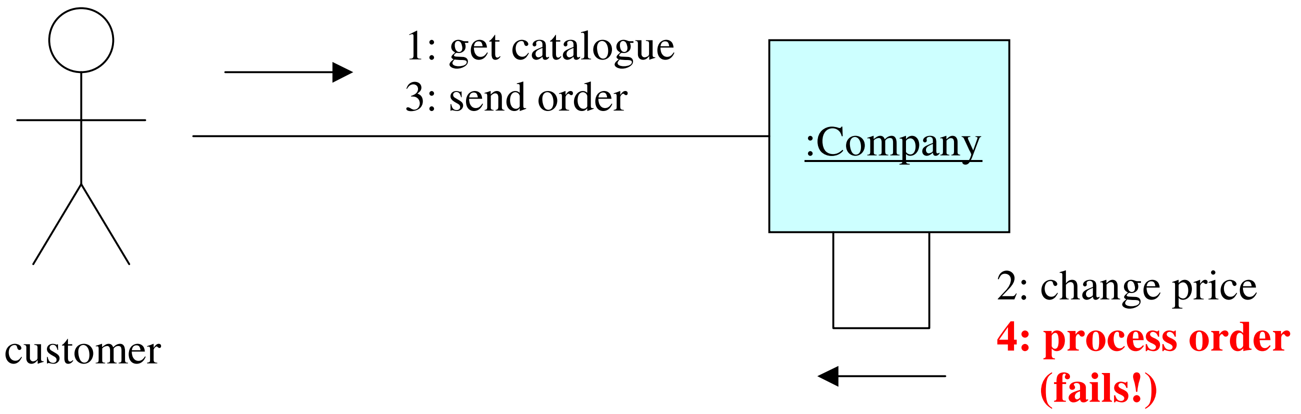
threads

serialisation

“Herb’s words”

solution space

Concurrency requirement example



- No threads, databases or locks involved
- It's not a technology problem
  - And technology can't solve it, either
  - The problem is in the real world, not in the "box"
- User gets a "surprise" - hidden assumptions

Other examples

- Tell a company that you've moved and they still sends bills to the old address
- Spouses accessing ATMs simultaneously
- Out-of-date documentation
- Bank account reconciliation
- Trying to arrange to meet friends in a pub
- Therac-25
- E-bay, CVS, etc, etc, etc…..

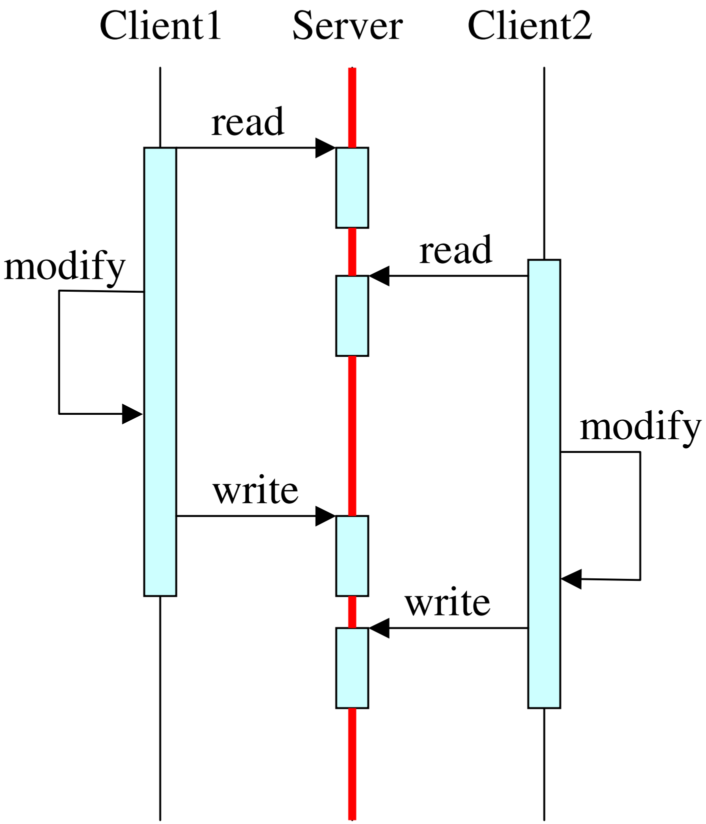
Solution to catalogue problem

- Catalogue has an expiry date
  - The company promises not to change prices for a given period (the lifetime of the catalogue)
- An example of a ***guarantee***
- Customer ***relies*** on this ***guarantee***
  - Without it, the transaction *may* fail
  - Hard to test thoroughly - non-deterministic
  - Implicit assumption of how things work
- An example of temporal logic
  - A time-based ***contract***

What are we trying do?

- We don’t have an effective way of defining this “contract”
  - We end up solving “something”
  - We may not be solving the “real” problem
- Could “Design by Contract™” help?
  - State the problem or define the solution
  - Conceived by Cliff Jones (1980 book)
  - Trademarked by Bertrand Meyer…

Design by Contract doesn't help



- DbC contracts relate to single operations only
  - All server contracts fulfilled
  - Point in time
- Client cannot rely on server
  - no temporal guarantees offered
- Invariant is true only when nothing is happening (red)!
  - During operation all bets are off
  - Re-entrancy not handled
- Classic lost-update problem

What clients need

- Guarantee between operations (time span)
  - This is the "hidden assumption"
- Can be stated in logic
  - ***rely*****:** catalogue price fixed throughout period
  - Testable mechanically but only at instants
- Server may offer such ***guarantees***
- If not, there is a potential race condition
  - Whenever ***guarantees*** do not span ***rely*** clauses

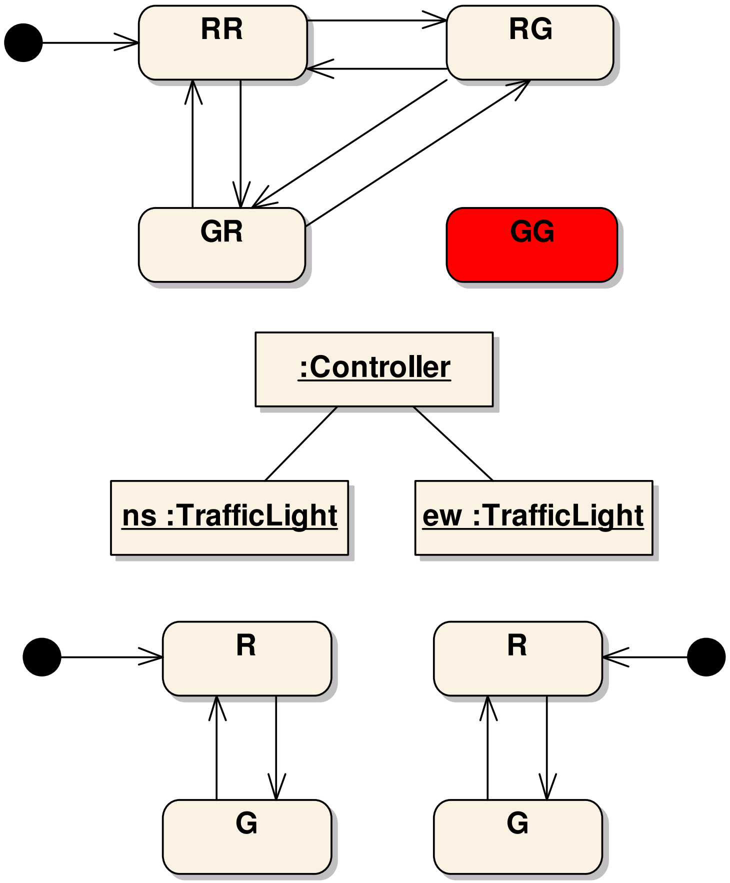
Catalogue guarantee

- Company must guarantee price does not change during period (problem)
- How? (Solution)
- And what should happen if someone tries?
  - Add surcharge, inform customers, stop trading?
- Example of an exclusive guarantee
  - Akin to locking or acquire/release
- (Still no technology involved….yet)

Time passes…

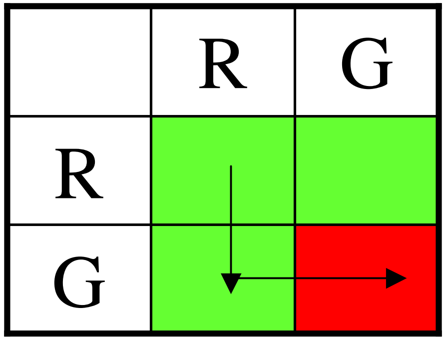
- After teaching much UML
  - many explanations of statecharts and hierarchical composition
  - talking to developers about analysis and requirements (problems v. solutions)
  - and many discussions with Derek Andrews (guardian of Catalysis and colleague of Cliff Jones)…
- …a thought struck me….

Concurrency and state machines



- Traffic light controller enforces a business rule: no more than one green
- Constrains the Cartesian product of the sub-state machines
- Centralised control of rules and fairness
- Top-down view

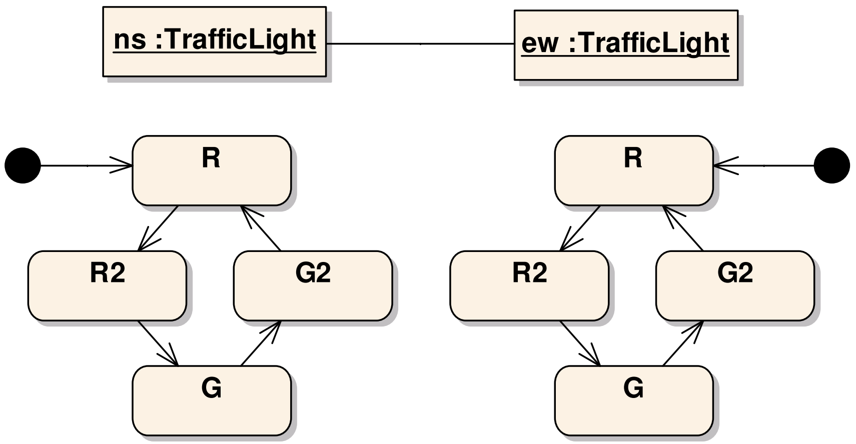
Composite state tables



- Cartesian product of states can be shown in tabular form
- One column and row per state of sub-state machines
- Events/transitions are on edges of boxes
- Path shows trace of events
- Diagonal (simultaneous) moves not allowed - race conditions

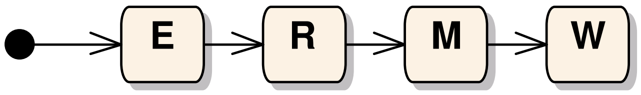
Peer-to-peer state machines

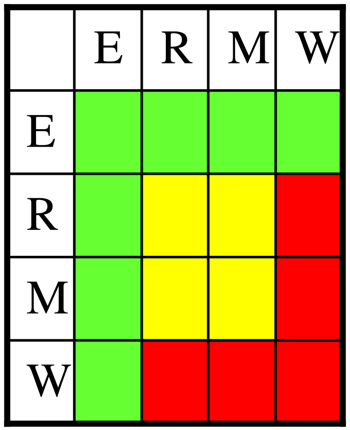
	R	R2	G	G2
R				
R2				
G				
G2				

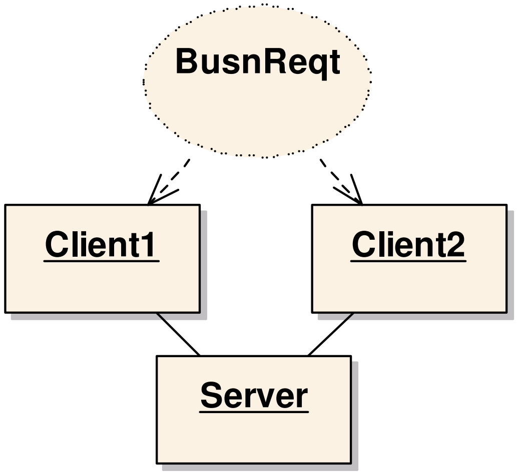


- Two traffic lights with no controller
- Same business rule
- Same “virtual” state machine (extra states for “fairness by timeout”)
- Enforcement by cooperation

Non-communicating clients



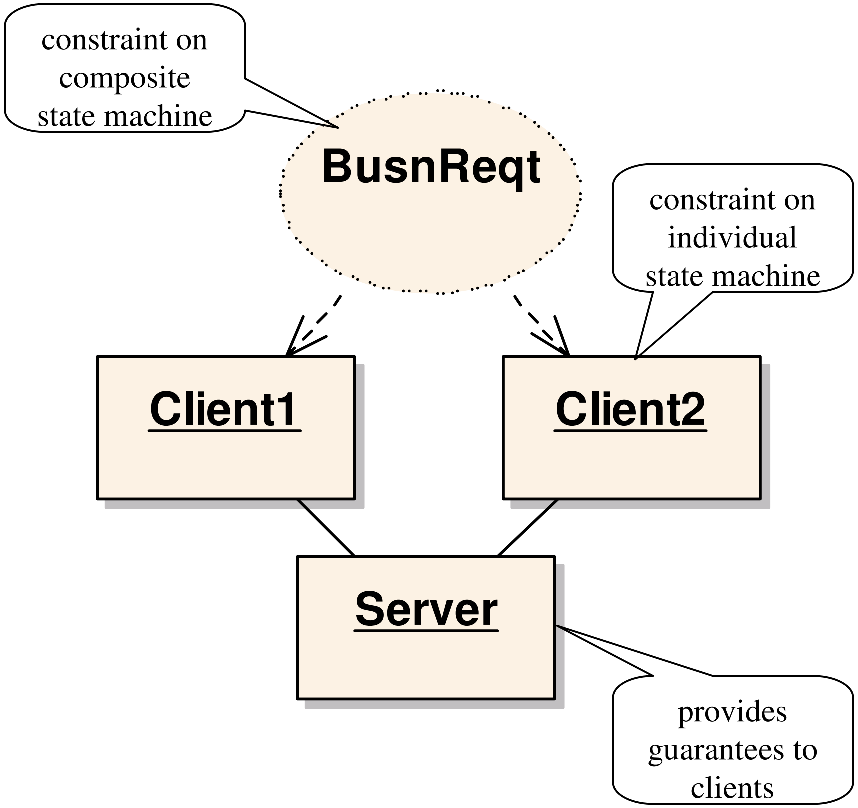




E = exists M = modified
R = read W = written

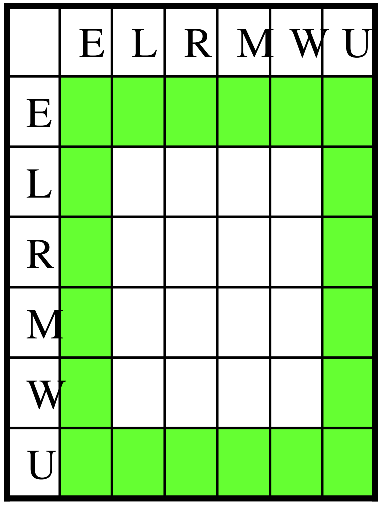
- Clients must synchronise via the server to enforce the business reqt on the composite state table
- “Control from below”

Layering of constraints

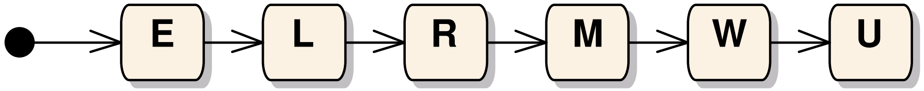


- Business reqt is virtual (no overall controller)
- Client requires cooperation of server to fulfil its part of the overall constraint ("relies" on server)
- Server's cooperates in the form of "guarantees"
  - Compensates for non-determinism - hiding
- "Signalling" via server
- Views and projections
- C.f. Michael Jackson

Pessimistic locking



E = exists
M = modified
L = locked
W = written
R = read
U = unlocked

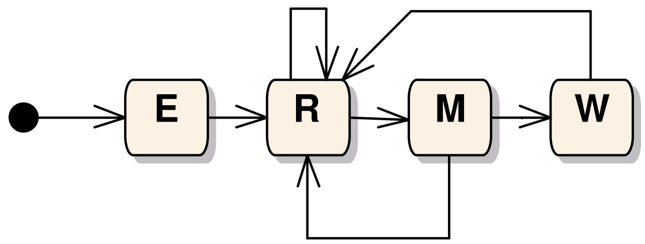


- Guarantee by exclusion
- Use an application-level lock to prevent unwanted states
- Spans multiple server-side operations
- Serialised access

Time-based guarantees

- If locking is not possible or desirable the server may have to guarantee that it will prevent certain operations during a given time period
  - e.g. Web page expiry times, catalogue problem
  - Certain events/operations will be refused
- Guarantees required so that client state machines don’t get into “unsafe” states
  - Composite state machine constraint satisfied
- Eliminates direct client communication
  - Indirect via server, provably correct

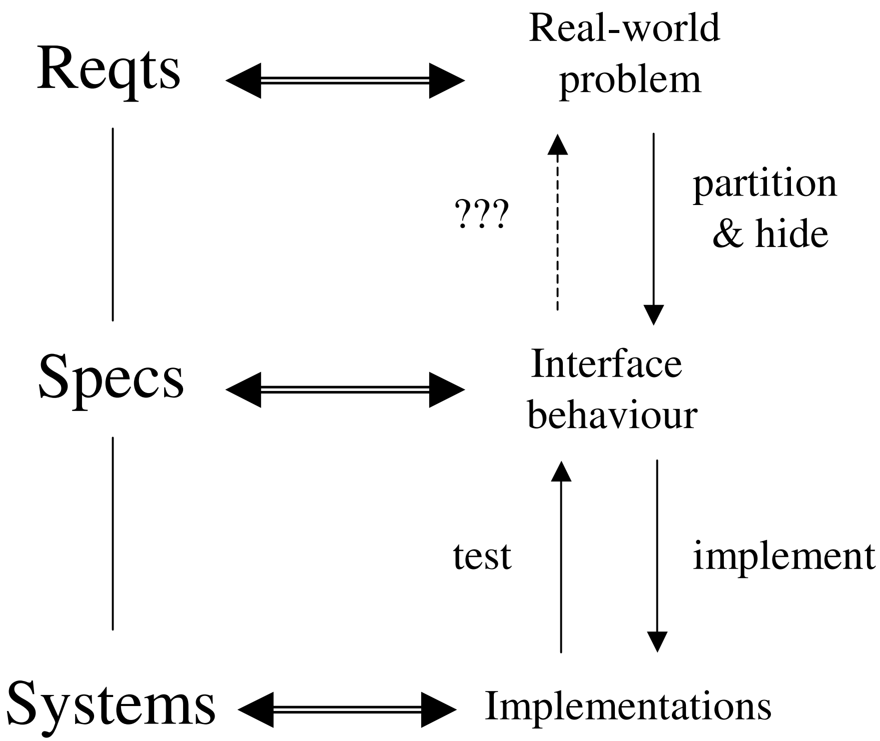
Notifications (signalling)



- Extra transitions in client state machine
- C.f. “unsafe” states (yellow and red)
- Resynchronisation is design problem/decision

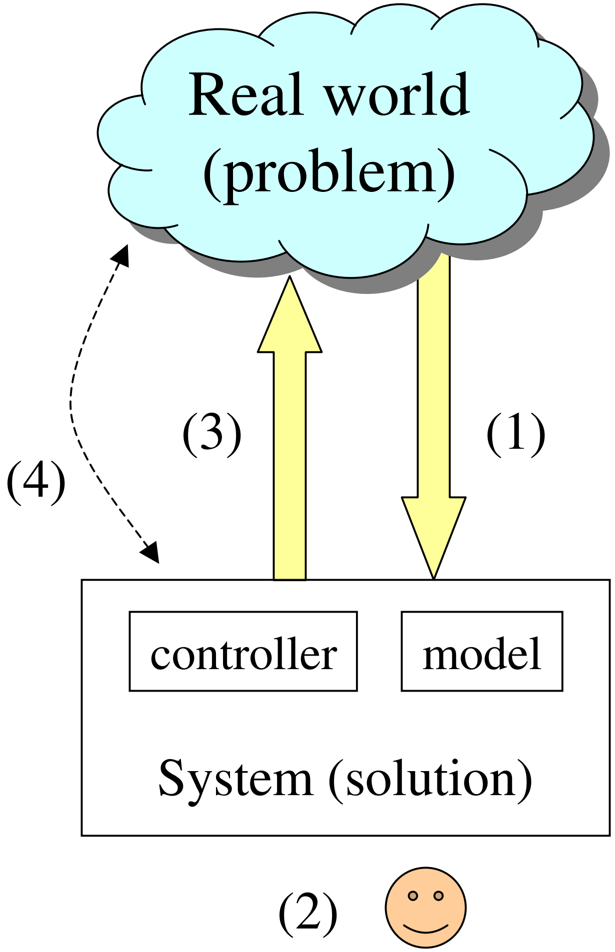
- Server communicates composite state machine events to clients
  - Interrupt (callback, Observer pattern, veto)
  - Next server operation (return code, sockets)
  - Client-side polling (delay!)
  - At end (optimistic “locking”, “lock free”)

Reqts and specs



- Partitioning and hiding can create non-deterministic specs
  - hard to test
- Recovering reqts from specs is hard
- TDD doesn’t help with concurrency
- Neither does DbC
  - Both are spec-based and deterministic

The problem is not in the box



1) Sufficient inputs to "see" problem (sensors)
2) Build deterministic model of the problem, not the i/f
3) Influence or control real world
4) Keep model in sync with real world (resynchronisation?)

- Responses in (3) may appear non-deterministic (hiding)
  - Rely and guarantee clauses eliminate non-determinism of i/f
- Use cases, TDD and DbC focus on (1) and (3), not the real world

Questions for the audience

- Is the problem concurrent or the solution?
- What real-world problem is your system trying to influence or control? How could you model it?
- Does your system have sufficient inputs to be able to achieve this, to enforce the rules or policies?
- Are some of your application problems concurrency in disguise (non-determinism)?
- How do you keep your system and the real world synchronised? What if they diverge?