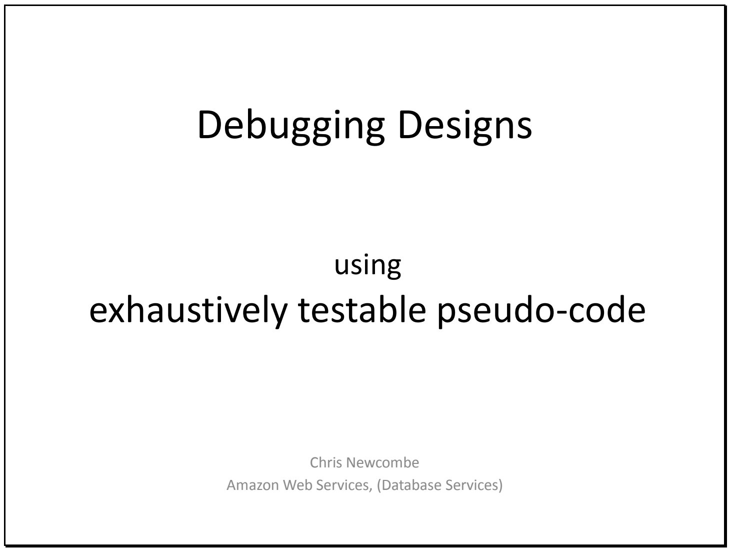
Slide 1



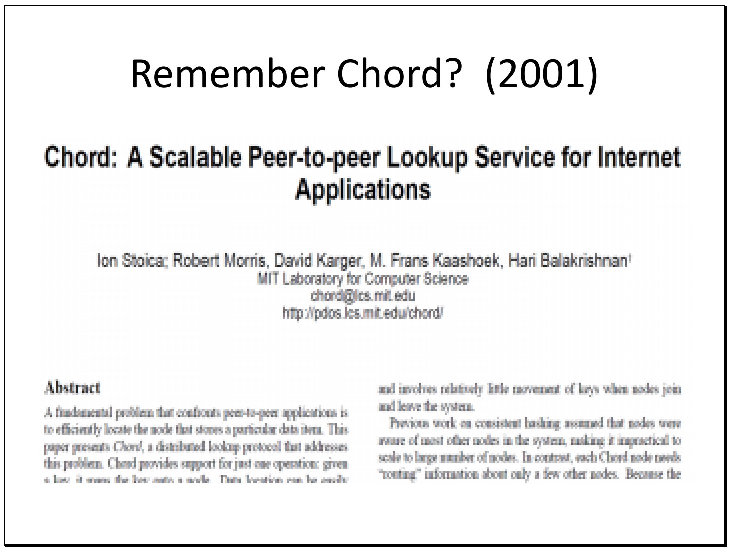
Debugging Designs

using
exhaustively testable pseudo-code

Chris Newcombe
Amazon Web Services, (Database Services)

30s
Amazon strives for services that are *simple to use*, but to meet our goals, the internals are often complex.
I'm going to talk about some tools we've been using to find subtle bugs in complex system designs.
<CLICK> Key idea: **exhaustively testable pseudo-code**

- Good programmers can learn to do this in their spare time over a couple of weeks.
- Has found problems that we might not have found by testing the implementation.
  e.g. It helped find a serious bug that requires 5 nodes and 34 steps (interactions/failures).
- Tools are mature and free



Remember Chord? (2001)

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica; Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan[†]
MIT Laboratory for Computer Science
chord@lcs.mit.edu
http://pdos.lcs.mit.edu/chord/

Abstract

A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item. This paper presents *Chord*, a distributed lookup protocol that addresses this problem. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily

and involves relatively little movement of keys when nodes join and leave the system.

Previous work on consistent hashing assumed that nodes were aware of most other nodes in the system, making it impractical to scale to large number of nodes. In contrast, each Chord node needs "routing" information about only a few other nodes. Because the

20s
According to citeseer, the **3rd most cited paper in computer science**
http://citeseerx.ist.psu.edu/stats/articles;jsessionid=9B4212E42789A894069928BDD698 78AC
Won **sigcomm 2011 "test of time award"**
http://www.sigcomm.org/awards/test-of-time-paper-award
Scrutinized far more than most designs.

Slide 3

Ring of nodes

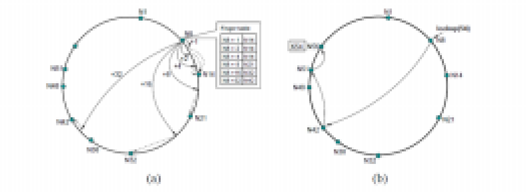
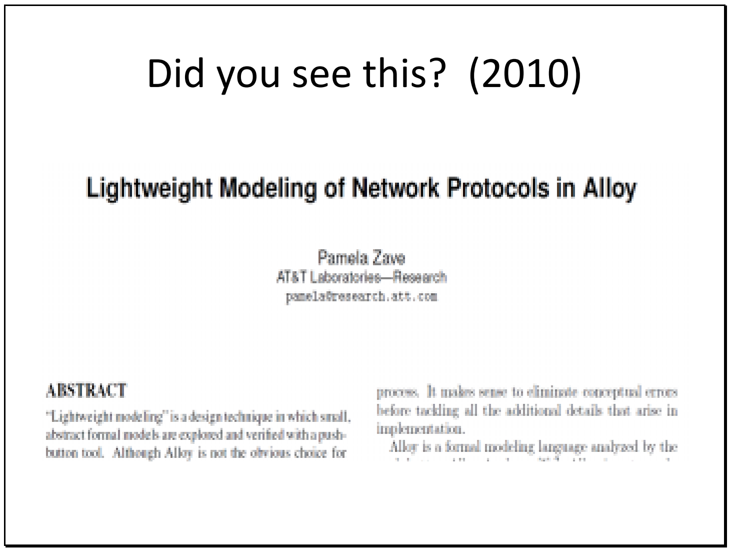


Fig. 4. (a) The finger table entries for node 8. (b) The path a query for key 54 starting at node 8, using the algorithm in Figure 5.

“Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, ***provable correctness,*** *and provable performance.”*

20s
Chord is a dynamic ring of nodes, with an O(log N) overlay network
<CLICK>
Can be used to build distributed hash-table key-value stores, which became the mainstay of the NoSQL movement.
<CLICK>
The paper emphasizes simplicity and correctness
Informal proofs for Chord are in MIT TR 819



Did you see this? (2010)

Lightweight Modeling of Network Protocols in Alloy

Pamela Zave
AT&T Laboratories—Research
pamela@research.att.com

ABSTRACT

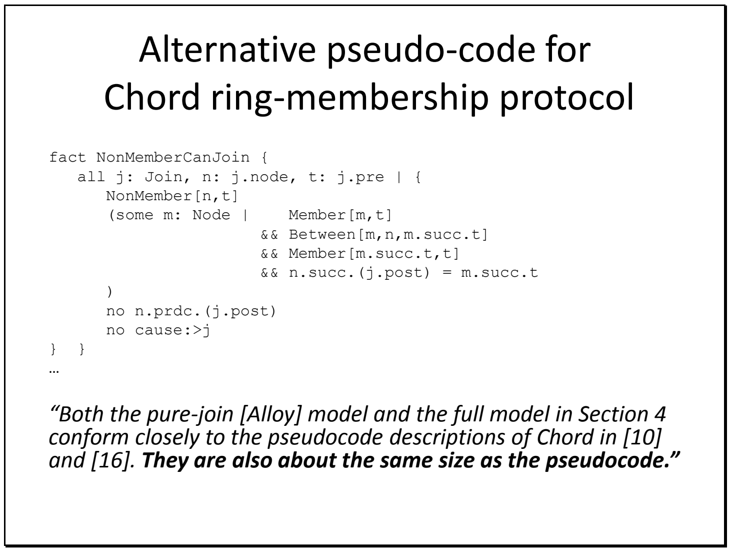
"Lightweight modeling" is a design technique in which small, abstract formal models are explored and verified with a push-button tool. Although Alloy is not the obvious choice for

process. It makes sense to eliminate conceptual errors before tackling all the additional details that arise in implementation.

Alloy is a formal modeling language analyzed by the

20s
Not even listed by citeseer
Might have got more attention if it was called <span style="color:red">**"How to quickly find major defects in your system design"**</span>

Pamela Zave works at AT&T, specifying and verifying telecoms protocols and switches.
Homepage including her pseudo-code of Chord ('Alloy models'):
http://www2.research.att.com/~pamela/model.html



Alternative pseudo-code for Chord ring-membership protocol

```
fact NonMemberCanJoin {
   all j: Join, n: j.node, t: j.pre | {
      NonMember[n,t]
      (some m: Node |    Member[m,t]
                      && Between[m,n,m.succ.t]
                      && Member[m.succ.t,t]
                      && n.succ.(j.post) = m.succ.t
      )
      no n.prdc.(j.post)
      no cause:>j
}  }
…
```

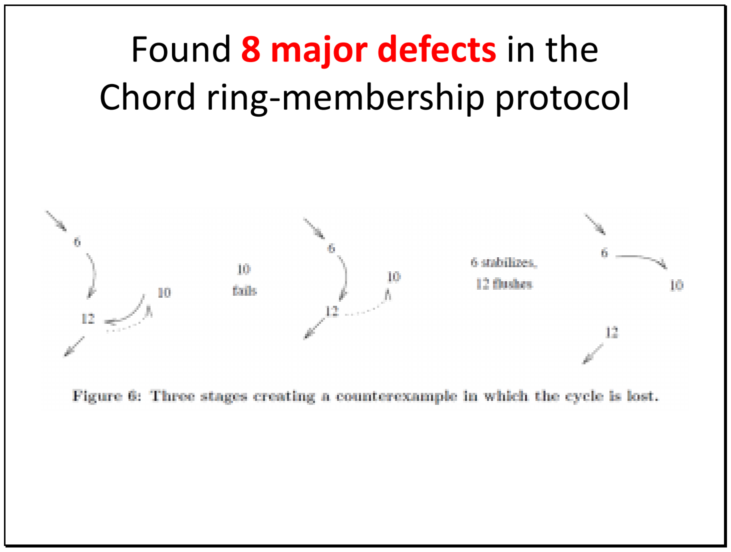
"Both the pure-join [Alloy] model and the full model in Section 4 conform closely to the pseudocode descriptions of Chord in [10] and [16]. ***They are also about the same size as the pseudocode."***

30s
Simple pseudo-code – any good programmer can write this.
Declarative : defines the legal state-space of the system.

<CLICK> same size as pseudo-code in original Chord paper

The Alloy language is simple predicate logic over relations of atomic identifiers.
  'fact' means this is an assumption about the model (part of the algorithm, not a correctness property or a helper function)
  a variable holds a relation of atoms – i.e. a set of tuples, all of which have the same number of elements (arity).
    A relation of arity 1 can be used as a set.
    A set with a single member can be used as a scalar.
  'all' is "for all" – i.e. universal quantification
  'some' is "there exists at least one" – i.e. existential quantification.
  'no' means "is the empty relation"
  '.' is relational join on the ends of tuples (to the left or right, depending on placement)
  ':>' is a filtering operation
  'X[y]' is a function call
An 'alloy model' is a single large constraint over a set of variables. The alloy analyzer can find an instances of all of the variables that satisfy the constraint (plus some 'interesting condition'), or help understand why the constraint cannot be satisfied.

Slide 6


Found **8 major defects** in the Chord ring-membership protocol

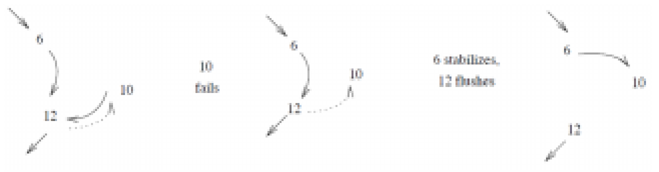


Figure 6: Three stages creating a counterexample in which the cycle is lost.

30s
Using the tool (Alloy Analyzer) on the new pseudo-code, they found defects.
The protocol cannot heal these scenarios – i.e. they are holes in the system design.

I'll show a few examples. Won't describe them in detail as this talk is about the technique used to find the problems, not about Chord.
<CLICK> a node fails at a certain point while it is trying to join, and breaks the ring.

Slide 7

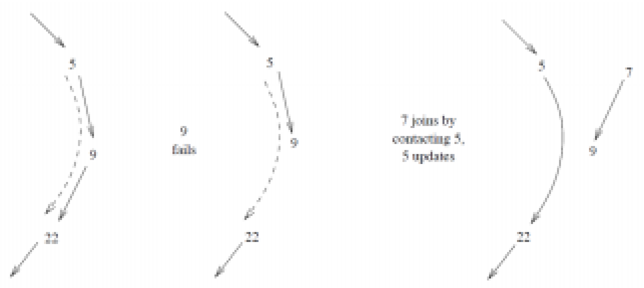
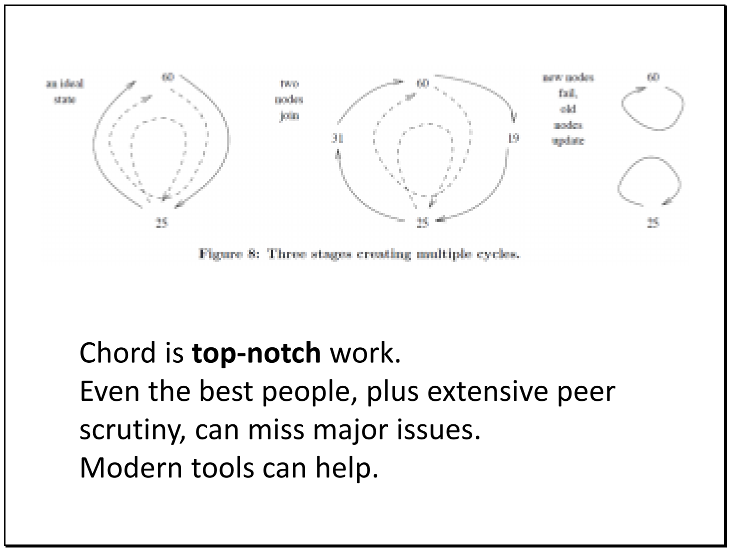


Figure 5: Three stages creating a counterexample to ConnectedAppendages.

10s
A particular failure while a node is trying to join means that the node doesn’t actually join

Slide 8



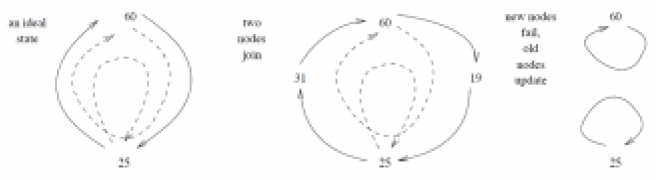
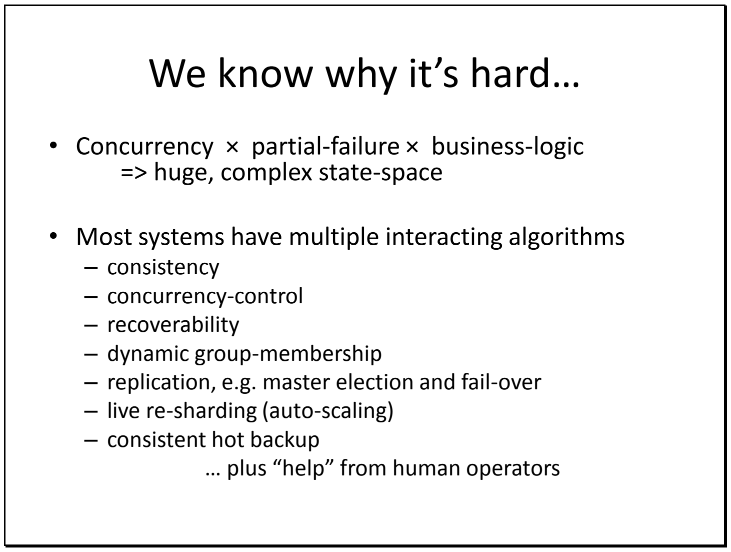


Figure 8: Three stages creating multiple cycles.

Chord is **top-notch** work.
Even the best people, plus extensive peer scrutiny, can miss major issues.
Modern tools can help.

40s
Particular failures while multiple nodes are joining result in multiple rings.
Other failures result in disordered rings.
<CLICK> for key point

Slide 9

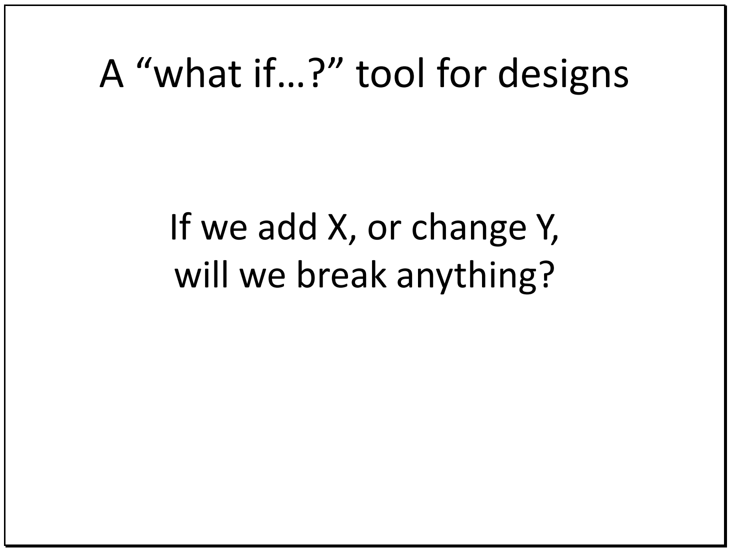


We know why it's hard…

- Concurrency × partial-failure × business-logic
  => huge, complex state-space
- Most systems have multiple interacting algorithms
  - consistency
  - concurrency-control
  - recoverability
  - dynamic group-membership
  - replication, e.g. master election and fail-over
  - live re-sharding (auto-scaling)
  - consistent hot backup

  … plus "help" from human operators

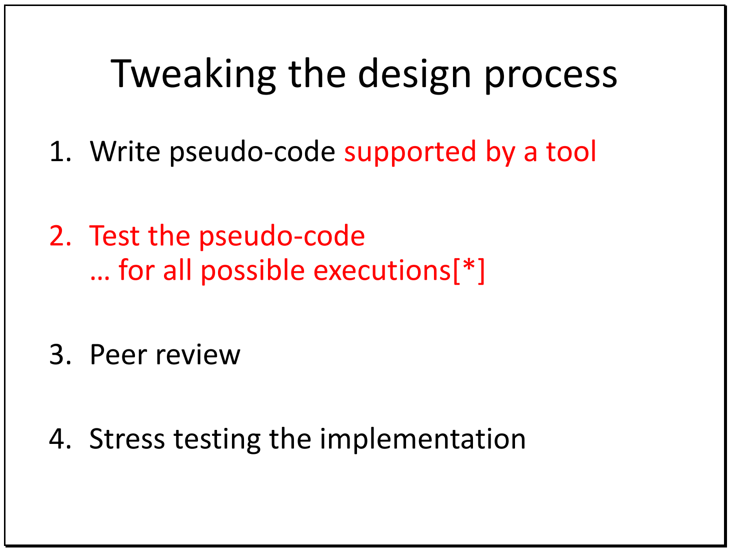
40s
Puny Earthling brains are poor at imagining permutations of execution-order interacting with partial-failure modes that vary over time.
<CLICK>
Note: *Chord is a relatively simple system* – it might be just one component of a modern system



A “what if…?” tool for designs

If we add X, or change Y,
will we break anything?

Slide 11



Tweaking the design process

1. Write pseudo-code supported by a tool
2. Test the pseudo-code
   … for all possible executions[*]
3. Peer review
4. Stress testing the implementation

1m20s
Taking the lessons from the Alloy paper…
Precise pseudo-code

No longer an ad-hoc language, e.g. precise pseudo-code **allows (requires) *complete control* over concurrency boundaries.**

Simple: want to be thinking about the problem-domain, not trying to remember language rules.

Helps you think by preventing hand-waving

Design reviews still important -- might catch something you forgot - e.g. some type of partial-failure or concurrency.

“Test all possible executions” : **for just a few objects**

In the famous paper, “*A Critique of ANSI SQL Isolation Levels*”, the various anomalies involve 2-3 transactions, 2-3 keys and key-versions, and system histories shorter than about 12 steps.

This is the key difference with modern high-level programming languages.

By “stress testing” I mean any measures to try to make edge-cases more likely to occur – e.g. fault-injection, concurrency-injection, overload to destruction. This is slow, expensive and not terribly effective.

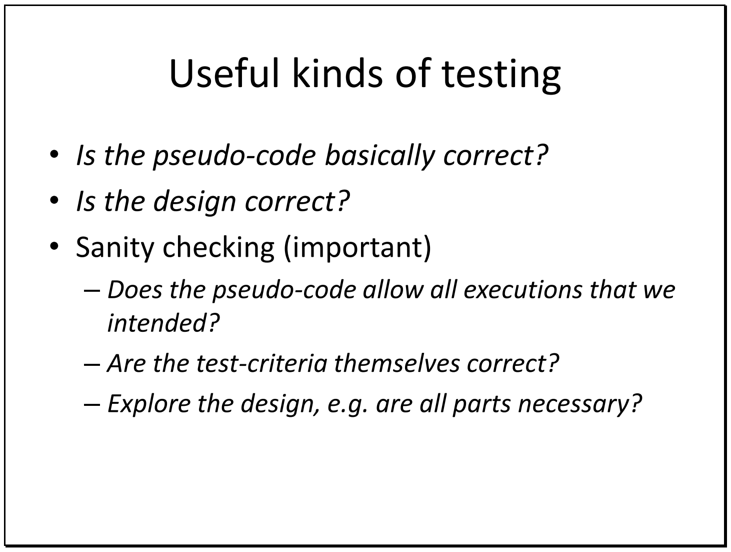
In *extreme* cases, e.g. safety-critical systems, we could also do:

3.5 Formal hierarchical proof

This is *enabled* by having this kind of precise pseudo-code, so they go hand-in-hand.

See Lamport: “How To Write A Proof”
“A method of writing proofs is proposed that makes it much harder to prove things that are not true.”
http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-how-to-write.pdf
Per Lamport’s paper, informal proof seems to be too error-prone to add any significant amount of confidence.
But test the pseudo-code first
Lamport: “it's a lot easier to prove something if it's true”

Slide 12



Useful kinds of testing

- *Is the pseudo-code basically correct?*
- *Is the design correct?*
- Sanity checking (important)
  - *Does the pseudo-code allow all executions that we intended?*
  - *Are the test-criteria themselves correct?*
  - *Explore the design, e.g. are all parts necessary?*

Does the code allow all interesting executions, or is it "accidentally over-constrained"?
- **very important** because it's an easy mistake to make, and can silently fool you into thinking that the design is correct. In the extreme, if no operations are allowed then all of the 'correctness of design' tests will still pass.
- E.g. I checked that each part of Cahill's algorithm does actually abort transactions to preserve serializability.

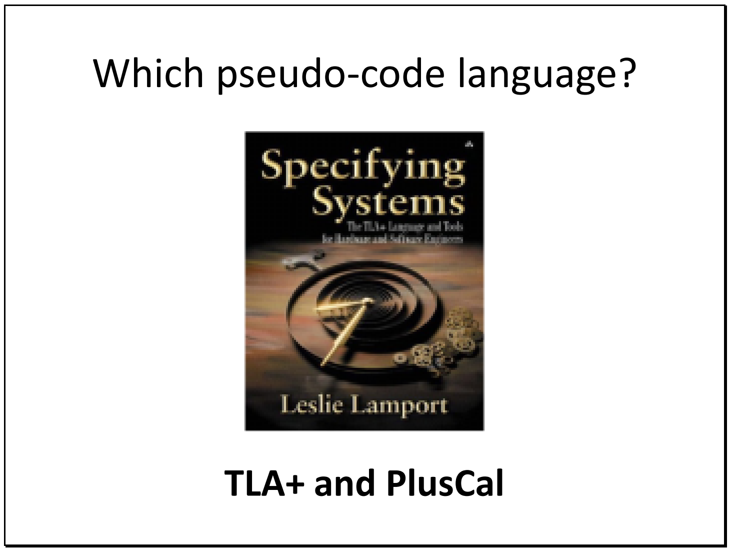
Are the tests themselves correct?
- E.g. I checked that different formulations of the correctness properties are equivalent.

Explore the design:
- E.g. I disabled parts of the algorithm (aborts intended to preserve serializability), and verified that correctness is violated as I expected.

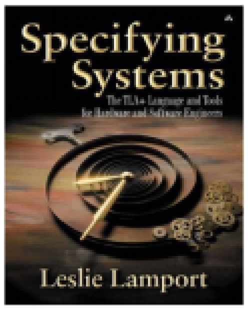
To begin with, focus on correctness properties that refer to a single current state : "invariant safety properties".
TLA+ can also express temporal properties – e.g. 'always', 'eventually', 'leads-to', weak and strong fairness etc. Don't worry too much about those to begin with. The fabled "Eventual Consistency" is a such temporal property.

See Lamport on "safety properties" and "liveness properties".

Slide 13



Which pseudo-code language?


Specifying Systems
The TLA+ Language and Tools for Hardware and Software Engineers
Leslie Lamport

TLA+ and PlusCal

3 min.
There are many, each one a PhD thesis
I've tried two; Alloy and TLA+/PlusCal.
I recommend TLA+/PlusCal.

We've been using TLA+ for real projects. We doubt that it is feasible to use Alloy to describe these systems, as they systems involve operations on non-trivial, rich data structures.

- Alloy is limited to relations over identifiers; **no data structures, no recursion.**
    - Makes it very fast to check small models (SAT solver). Interactive edit-check-debug.
    - The current SAT solvers usual crash on realistic models.

- TLA+ is much more expressive.
    - TLA+ is slower to check for small scopes. But now supports distributed model checking.
    - TLA+ includes a semi-automated hierarchical proof system
    - **PlusCal is more conventional syntax; assignment, loops. Still with explicit control over concurrency.**

Examples of TLA+ in industry

Farsite distributed file system (byzantine-fault tolerant directory server)
http://research.microsoft.com/apps/pubs/default.aspx?id=74211
Alpha SMP cache-coherency

http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#fm99

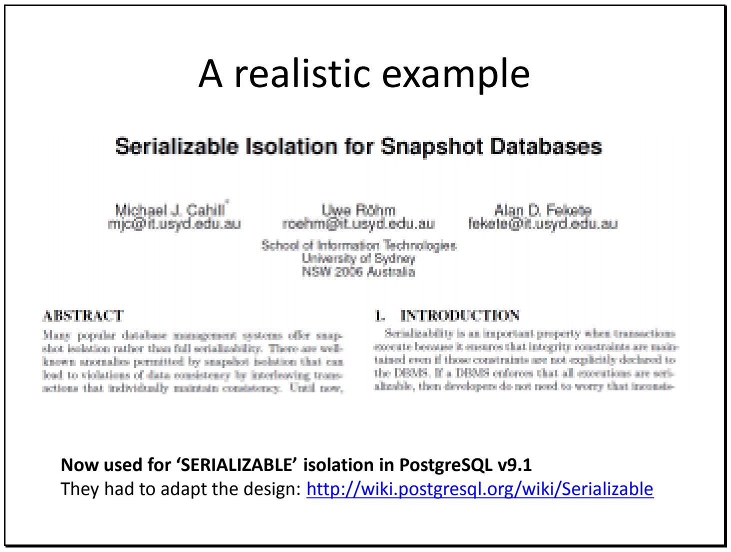
Intel:

http://research.microsoft.com/pubs/64638/high-level.pdf
page 4 of http://research.microsoft.com/en-us/um/people/lamport/pubs/commentary-web.pdf

Security

http://profsandhu.com/it862/it862s05/ucon-tla1.pdf

Slide 14



A realistic example

Serializable Isolation for Snapshot Databases

Michael J. Cahill[*] mjc@it.usyd.edu.au
Uwe Röhm roehm@it.usyd.edu.au
Alan D. Fekete fekete@it.usyd.edu.au
School of Information Technologies
University of Sydney
NSW 2006 Australia

ABSTRACT

Many popular database management systems offer snapshot isolation rather than full serializability. There are well-known anomalies permitted by snapshot isolation that can lead to violations of data consistency by interleaving transactions that individually maintain consistency. Until now,

1. INTRODUCTION

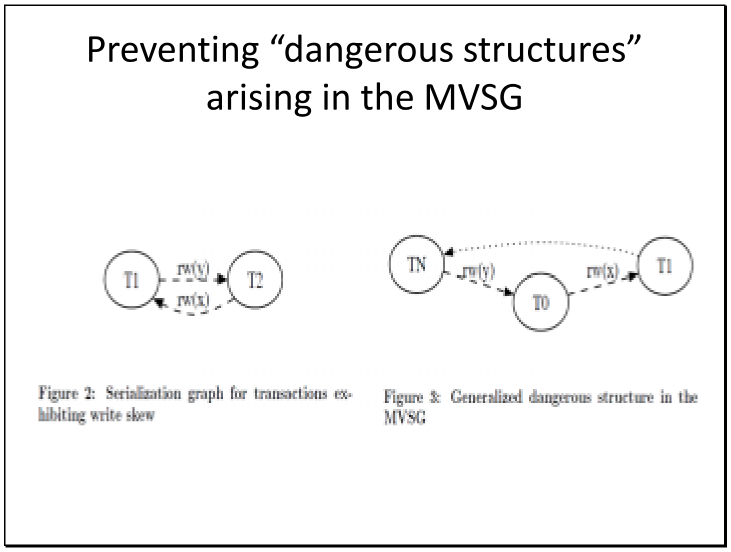
Serializability is an important property when transactions execute because it ensures that integrity constraints are maintained even if those constraints are not explicitly declared to the DBMS. If a DBMS enforces that all executions are serializable, then developers do not need to worry that inconsis-

Now used for 'SERIALIZABLE' isolation in PostgreSQL v9.1
They had to adapt the design: http://wiki.postgresql.org/wiki/Serializable

Best paper award at SIGMOD 2008

Practical, realistic : **<CLICK>**

Paper: http://cahill.net.au/wp-content/uploads/2009/01/real-serializable.pdf
PhD thesis: http://cahill.net.au/wp-content/uploads/2010/02/cahill-thesis.pdf



Preventing "dangerous structures" arising in the MVSG

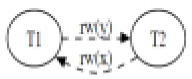


Figure 2: Serialization graph for transactions exhibiting write skew

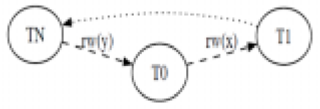
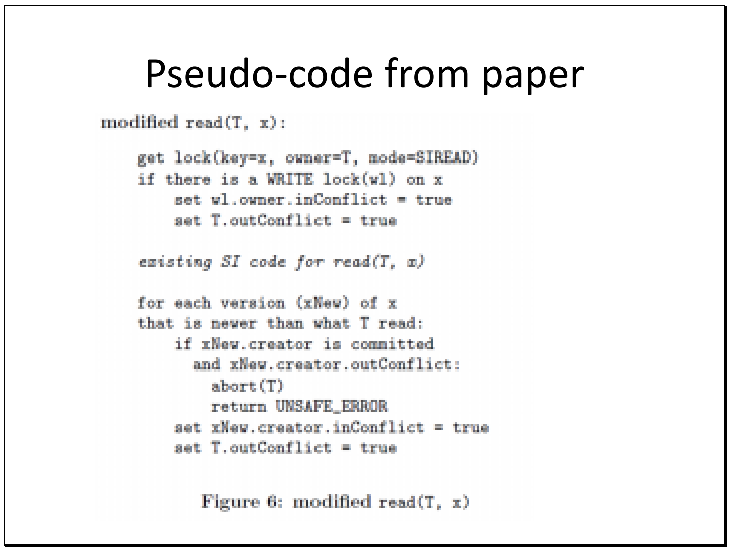


Figure 3: Generalized dangerous structure in the MVSG

Cahill's algorithm works by preventing the creation of cycles in the MVSG which contain two consecutive read-write dependencies between concurrent transactions. A "read-write-dependency" is: Tn reads a key and a different Tm writes a later version of that key.

A non-trivial correctness condition; transaction serializability over multi-valued objects

Slide 16



Pseudo-code from paper

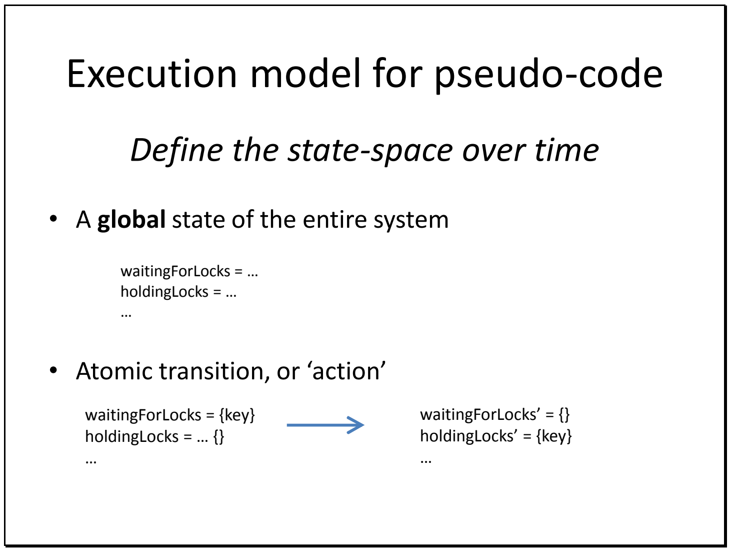
modified read(T, x):

```
get lock(key=x, owner=T, mode=SIREAD)
if there is a WRITE lock(wl) on x
    set wl.owner.inConflict = true
    set T.outConflict = true

existing SI code for read(T, x)

for each version (xNew) of x
that is newer than what T read:
    if xNew.creator is committed
      and xNew.creator.outConflict:
        abort(T)
        return UNSAFE_ERROR
    set xNew.creator.inConflict = true
    set T.outConflict = true
```

Figure 6: modified read(T, x)

Let’s make this exhaustively testable!

Slide 17



Execution model for pseudo-code

Define the state-space over time

- A **global** state of the entire system

```
waitingForLocks = …
holdingLocks = …
…
```

- Atomic transition, or ‘action’

```
waitingForLocks = {key}      ──→      waitingForLocks’ = {}
holdingLocks = … {}                   holdingLocks’ = {key}
…                                     …
```

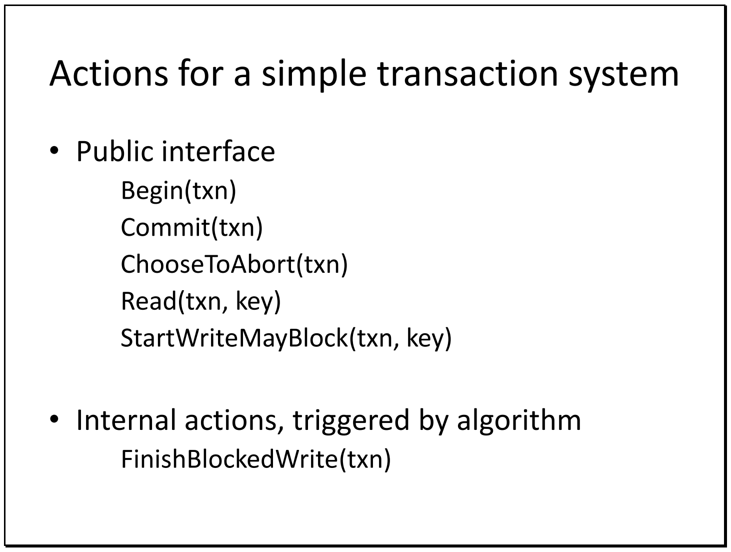
Counter-intuitive idea: the best way to model a system is **a single global state** with atomic transitions.
http://research.microsoft.com/en-us/um/people/lamport/pubs/state-machine.pdf

Declaratively define state-space : this enables tools to exhaustively explore it.

Just some variables with (rich) values.
e.g. For a distributed message passing system,
- One variable might model the current messages in transit. (Example transitions might be: lose a message, duplicate a message, change the order of messages.)
- Other variables would model internal process state

In theory, psuedo-code could atomically refer to variables representing state in different nodes. In practice you would choose to not do that, as it is impractical to implement such a design.

Must also choose the allowed **initial states** of the system – usually trivial.



Actions for a simple transaction system

- Public interface

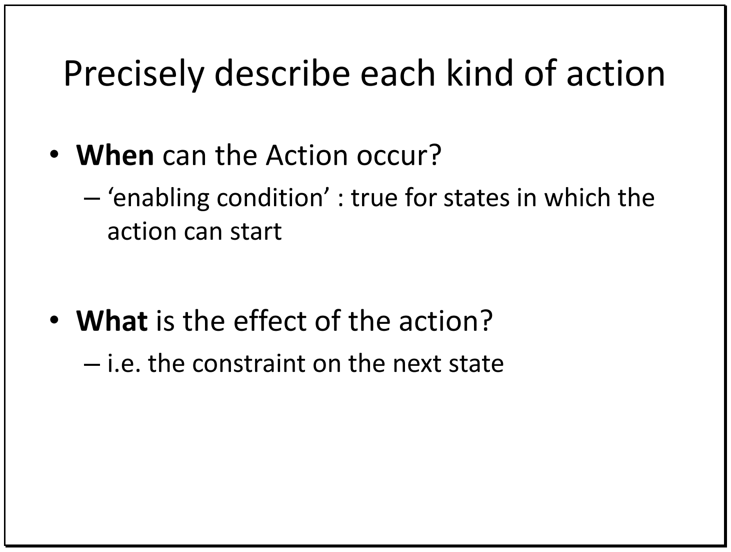
  Begin(txn)
  Commit(txn)
  ChooseToAbort(txn)
  Read(txn, key)
  StartWriteMayBlock(txn, key)

- Internal actions, triggered by algorithm

  FinishBlockedWrite(txn)

A distributed/fault-tolerant system would also have
  - Actions performed by the environment -- e.g. node failure, breaking a connection, losing a message, duplicating a message, changing order of two messages in transit
  - Actions performed by human operator

Slide 19



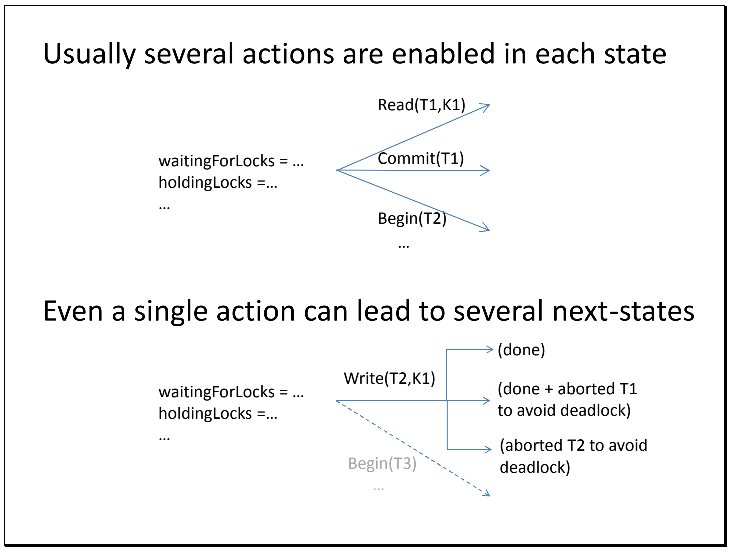
Precisely describe each kind of action

- **When** can the Action occur?
  - 'enabling condition' : true for states in which the action can start
- **What** is the effect of the action?
  - i.e. the constraint on the next state

Explicit granularity of concurrency
Declarative and expressive: first-order logic with quantifiers, set-comprehensions, recursive operators/functions.
Want to specify the *weakest* constraint on the next state.

The union (disjunction) of all of the actions is the formula for the 'next state action' of the system.

Slide 20

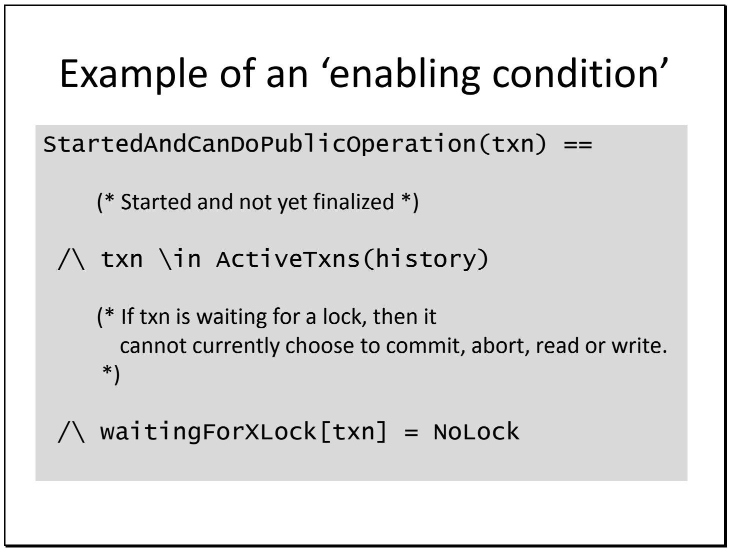


Usually several actions are enabled in each state

Even a single action can lead to several next-states

waitingForLocks = …
holdingLocks =…
…
Write(T2,K1)
(done)
(done + aborted T1 to avoid deadlock)
(aborted T2 to avoid deadlock)
Begin(T3)
…

Non-determinism. We want to exhaustively test ALL possible consequences (for some small number of transactions and keys).



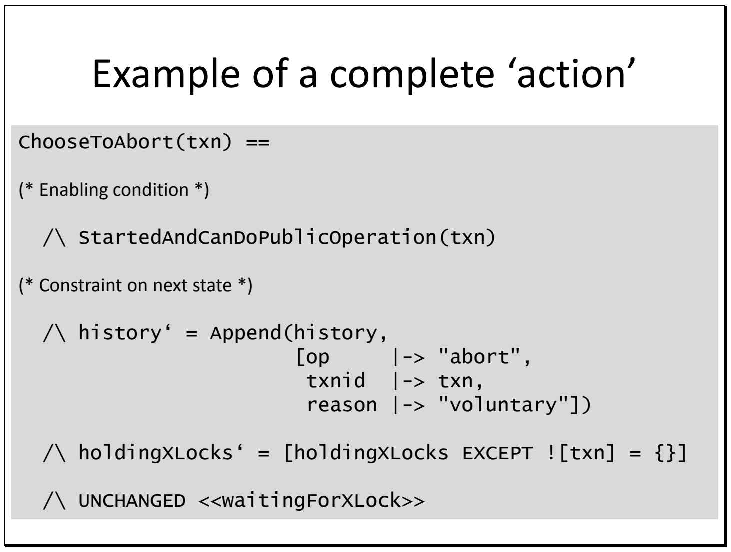
Example of an ‘enabling condition’

```
StartedAndCanDoPublicOperation(txn) ==

      (* Started and not yet finalized *)

 /\ txn \in ActiveTxns(history)

      (* If txn is waiting for a lock, then it
         cannot currently choose to commit, abort, read or write.
       *)

 /\ waitingForXLock[txn] = NoLock
```

This is the enabling condition for all of:

- Commit(txn)
- Read(txn, key)
- Write(txn, key)
- ChooseToAbort(txn)

Slide 22



Example of a complete ‘action’

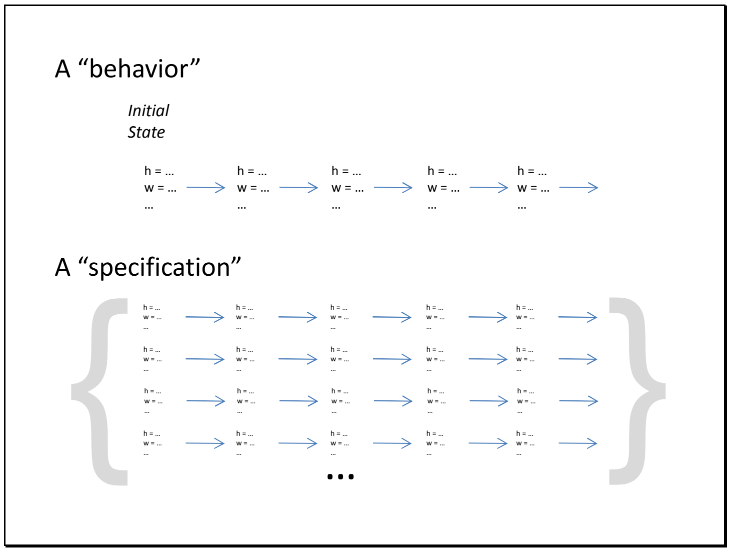
```
ChooseToAbort(txn) ==

(* Enabling condition *)

  /\ StartedAndCanDoPublicOperation(txn)

(* Constraint on next state *)

  /\ history‘ = Append(history,
                       [op     |-> "abort",
                        txnid  |-> txn,
                        reason |-> "voluntary"])

  /\ holdingXLocks‘ = [holdingXLocks EXCEPT ![txn] = {}]

  /\ UNCHANGED <<waitingForXLock>>
```

waitingForXLock does not change because we know that StartedAndCanDoPublicOperations(txn) is true, which means that txn cannot be waiting for a lock.

But it would not hurt to write:

```
    waitingForXLock ‘ = [waitingForXLock EXCEPT ![txn] = {}]
```

Slide 23


A "behavior"

Initial State

A "specification"

Some terminology **<CLICK>**

A *behavior* is a sequence of state-transitions (aka actions, steps).
One possible history of the universe.
Infinitely long.
Starts in one of the possible initial states

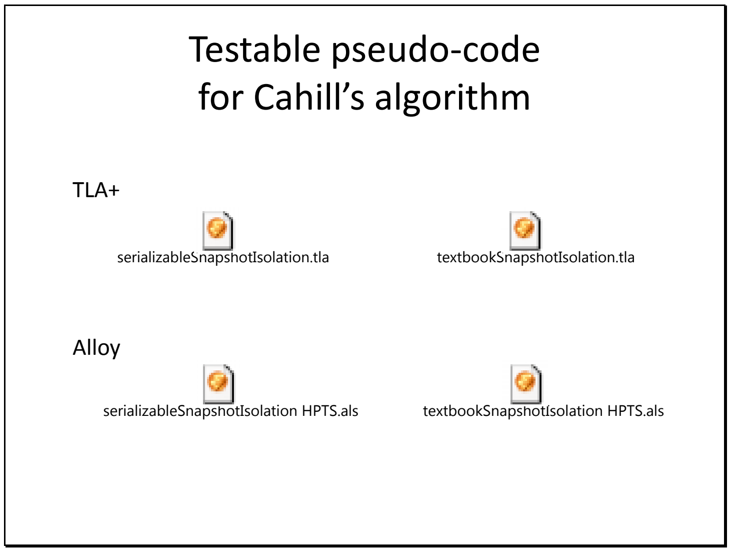
A *specification* is a set of allowed behaviors, possibly infinitely many.
Each behavior generated by choosing a different enabled action in some state

The pseudo code describes the specification
The pseudo code is a predicate ('if-statement') that for any possible behavior, evaluates to true if and only if that behavior is a legal execution of the system.
So, declarative pseudo-code 'recognizes' valid behaviors, rather that imperatively generating them.
Most of the pseudo-code deals with the current state (unprimed variables), and immediate next state (primed variables).

This sounds weird at first, but it "feels" just like writing code. It feels like the code is 'generating' the next state.
Takes a couple of weeks to learn.

The model-checker evaluates the correctness-criteria for all behaviors allowed by the specification,
(for some small finite number of objects).

Slide 24



Testable pseudo-code for Cahill's algorithm

TLA+

serializableSnapshotIsolation.tla

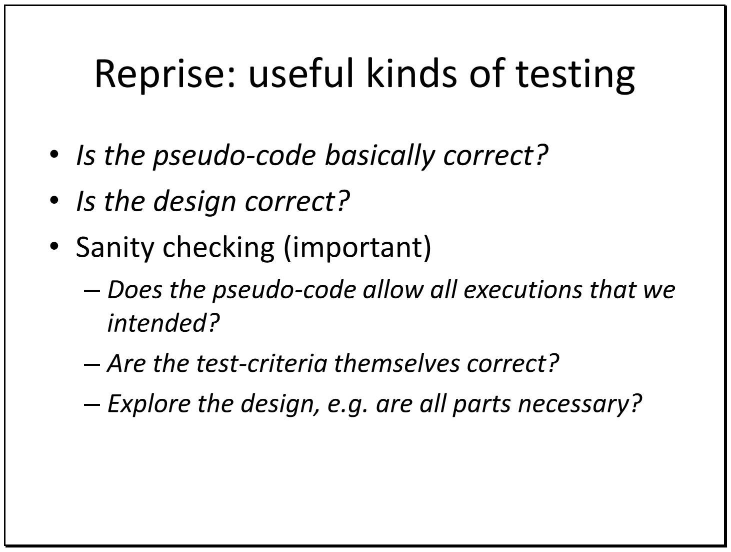
textbookSnapshotIsolation.tla

Alloy

serializableSnapshotIsolation HPTS.als

textbookSnapshotIsolation HPTS.als

These are ascii text files. To try running the 'exhaustive tests', follow the instructions in comments at the top of each file.
(To obtain the IDEs/tools, see the links in the last few slides of this talk.)



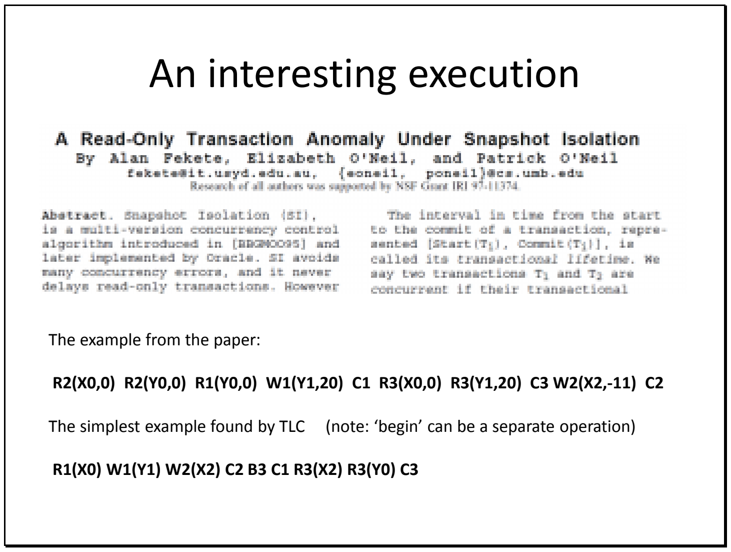
Reprise: useful kinds of testing

- *Is the pseudo-code basically correct?*
- *Is the design correct?*
- Sanity checking (important)
  - *Does the pseudo-code allow all executions that we intended?*
  - *Are the test-criteria themselves correct?*
  - *Explore the design, e.g. are all parts necessary?*

In the example TLA+ files:
Is it basically correct?
TypeInv, WellFormedTransactionsInHistory, CorrectnessOfHoldingXLocks, CorrectnessOfWaitingForXLock
Is the design correct?
CahillSerializable, FirstCommitterWins, CorrectReadView
Does the code allow all intended executions?
I checked that each part of Cahill's algorithm does actually abort transactions to preserve serializability.
Are the tests themselves correct?
I checked that different formulations of the correctness properties are equivalent.
CahillSerializable(history) = BernsteinSerializable(history
Explore the design:
I disabled parts of the algorithm (aborts intended to preserve serializability), and verified that correctness is violated as I expected.

To begin with, focus on correctness properties that refer to a single current state : "invariant safety properties".
TLA+ can also express temporal properties – e.g. 'always', 'eventually', 'leads-to', weak and strong fairness etc. Don't worry too much about those to begin with. The fabled "Eventual Consistency" is a such temporal property.
See Lamport on "safety properties" and "liveness properties".

Slide 26



An interesting execution

A Read-Only Transaction Anomaly Under Snapshot Isolation
By Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil
fekete@it.usyd.edu.au, {eoneil, poneil}@cs.umb.edu
Research of all authors was supported by NSF Grant IRI 97-11374.

Abstract. Snapshot Isolation (SI), is a multi-version concurrency control algorithm introduced in [BBGMOO95] and later implemented by Oracle. SI avoids many concurrency errors, and it never delays read-only transactions. However

The interval in time from the start to the commit of a transaction, represented [Start(T$_i$), Commit(T$_i$)], is called its transactional lifetime. We say two transactions T$_1$ and T$_2$ are concurrent if their transactional

The example from the paper:

R2(X0,0) R2(Y0,0) R1(Y0,0) W1(Y1,20) C1 R3(X0,0) R3(Y1,20) C3 W2(X2,-11) C2

The simplest example found by TLC (note: 'begin' can be a separate operation)

R1(X0) W1(Y1) W2(X2) C2 B3 C1 R3(X2) R3(Y0) C3

The tools can be used to search for "interesting conditions", such as this one described by Fekete et al.
(Instructions below and in the file textbookSnapshotIsolation.tla)

For textbookSnapshotIsolation.tla, TLC found the following example in 6 hours 21 minutes on an EC2 CC1 Windows instance.
(TLC config: Key = {K0, K1}, TxnId {T0, T1, T2, T3} with both configured as symmetry-sets of untyped model values)

```
… preamble to create first version of each key  (assumed by example in paper)
   [op |-> "begin", txnid |-> T0],
   [op |-> "write", txnid |-> T0, key |-> K0],
   [op |-> "write", txnid |-> T0, key |-> K1],
   [op |-> "commit", txnid |-> T0],
… the interesting part
   [op |-> "begin", txnid |-> T1],
   [op |-> "read", txnid |-> T1, key |-> K0, ver |-> T0],
   [op |-> "write", txnid |-> T1, key |-> K1],
   [op |-> "begin", txnid |-> T2],
   [op |-> "write", txnid |-> T2, key |-> K0],
   [op |-> "commit", txnid |-> T2],
   [op |-> "begin", txnid |-> T3],
  [op |-> "commit", txnid |-> T1],
  [op |-> "read", txnid |-> T3, key |-> K0, ver |-> T2],
  [op |-> "read", txnid |-> T3, key |-> K1, ver |-> T0],
  [op |-> "commit", txnid |-> T3] >>
```

Found by claiming ‘checking’ that this predicate is invariant, so that TLC reports the first state that violates it:

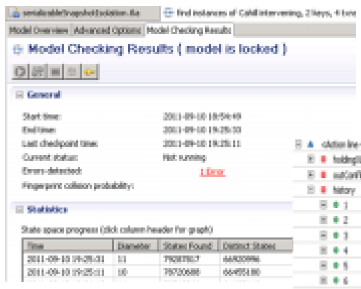
```
   ~ ReadOnlyAnomaly(history)
```

with:

```
ReadOnlyAnomaly(h) ==
        (* current history is not serializable … *)
    /\  ~ CahillSerializable(h)
        (* … and there is a transaction that does some reads and
zero writes,
               and, when that transaction is entirely removed
from the history, the resulting history is serializable. *)
    /\ \E txn \in TxnId :
            LET keysReadWritten == KeysReadAndWrittenByTxn(h,
txn)
            IN
                /\ Cardinality(keysReadWritten[1]) > 0
                /\ Cardinality(keysReadWritten[2]) = 0
                /\ CahillSerializable(HistoryWithoutTxn(h, txn))
```

Slide 27

TLC checking Cahill’s algorithm



serializableSnapshotIsolation.tla | Find instances of Cahill intervening, 2 keys, 4 txns

Model Overview | Advanced Options | Model Checking Results

Model Checking Results (model is locked)

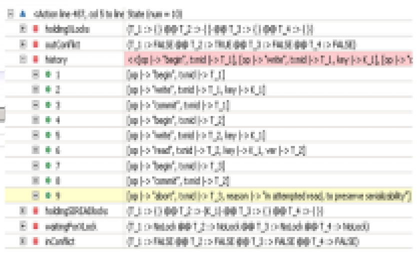
General

Start time:	2011-09-10 18:54:49
End time:	2011-09-10 19:25:33
Last checkpoint time:	2011-09-10 19:25:11
Current status:	Not running
Errors detected:	1 Error
Fingerprint collision probability:	

Statistics

State space progress (click column header for graph)

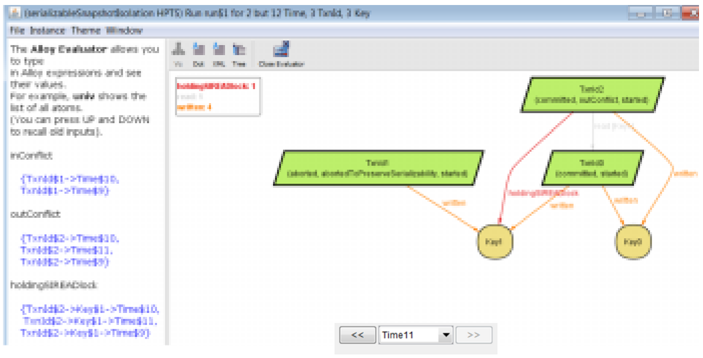
Time	Diameter	States Found	Distinct States
2011-09-10 19:25:31	11	79287817	66420996
2011-09-10 19:25:11	10	79720688	66495180



```
<Action line 487, col 5 to line   State (num = 10)
  holdingSILocks        (T_1 :> {} @@ T_2 :> {} @@ T_3 :> {} @@ T_4 :> {})
  outConflict           (T_1 :> FALSE @@ T_2 :> TRUE @@ T_3 :> FALSE @@ T_4 :> FALSE)
  history               <<[op |-> "begin", txnid |-> T_1], [op |-> "write", txnid |-> T_1, key |-> K_1], [op |-> "c
    1                   [op |-> "begin", txnid |-> T_1]
    2                   [op |-> "write", txnid |-> T_1, key |-> K_1]
    3                   [op |-> "commit", txnid |-> T_1]
    4                   [op |-> "begin", txnid |-> T_2]
    5                   [op |-> "write", txnid |-> T_2, key |-> K_1]
    6                   [op |-> "read", txnid |-> T_2, key |-> K_1, ver |-> T_2]
    7                   [op |-> "begin", txnid |-> T_3]
    8                   [op |-> "commit", txnid |-> T_2]
    9                   [op |-> "abort", txnid |-> T_3, reason |-> "in attempted read, to preserve serializability"]
  holdingSIREADlocks    (T_1 :> {} @@ T_2 :> {K_1} @@ T_3 :> {} @@ T_4 :> {})
  waitingForXLock       (T_1 :> NoLock @@ T_2 :> NoLock @@ T_3 :> NoLock @@ T_4 :> NoLock)
  inConflict            (T_1 :> FALSE @@ T_2 :> FALSE @@ T_3 :> FALSE @@ T_4 :> FALSE)
```

Slide 28

Alloy checking Cahill's algorithm



(serializableSnapshotIsolation HPTS) Run run$1 for 2 but 12 Time, 3 TxnId, 3 Key

The **Alloy Evaluator** allows you to type
in Alloy expressions and see their values.
For example, **univ** shows the list of all atoms.
(You can press UP and DOWN to recall old inputs).

inConflict

```
{TxnId$1->Time$10,
 TxnId$1->Time$9}
```

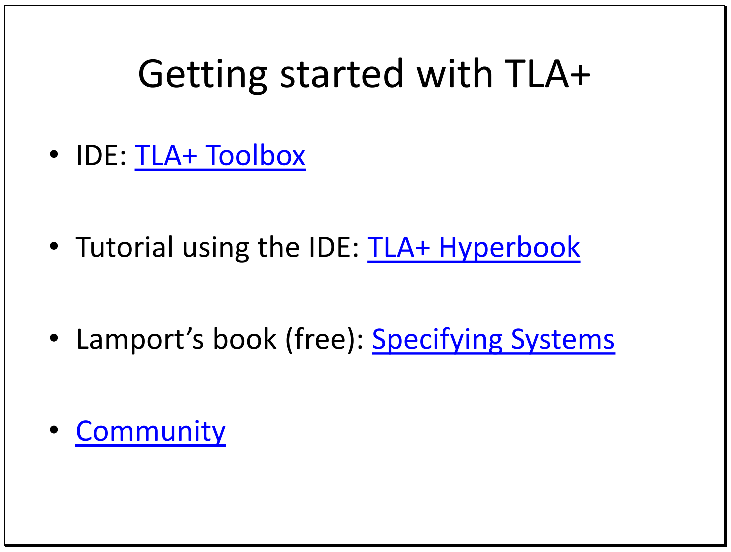
outConflict

```
{TxnId$2->Time$10,
 TxnId$2->Time$11,
 TxnId$2->Time$9}
```

holdingSIREADlock

```
{TxnId$2->Key$1->Time$10,
 TxnId$2->Key$1->Time$11,
 TxnId$2->Key$1->Time$9}
```

Slide 29



Getting started with TLA+

- IDE: TLA+ Toolbox
- Tutorial using the IDE: TLA+ Hyperbook
- Lamport's book (free): Specifying Systems
- Community

PlusCal is a more convention syntax for TLA+ : http://research.microsoft.com/en-us/um/people/lamport/tla/pluscal.html

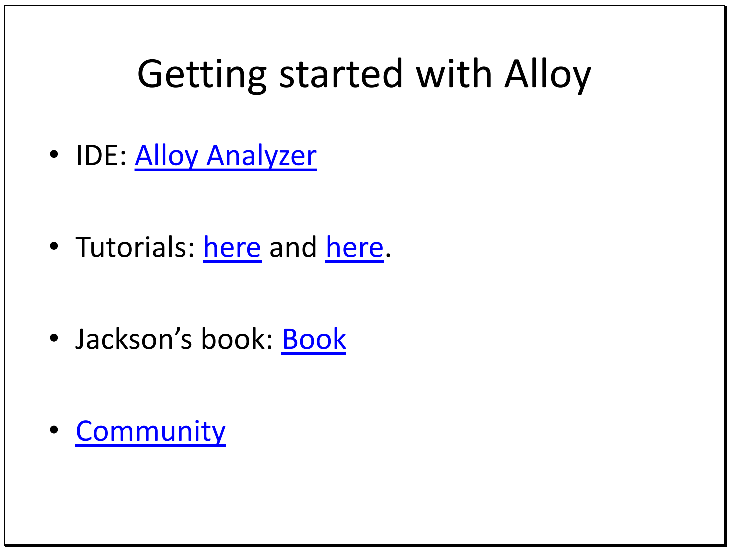
An anecdote from one engineer who took the above path. (He characterized the resulting model as "hugely valuable".)

"The actual time I spent learning enough TLA+ to get started wasn't much (it surprised me how easy it was to pick up enough to be dangerous.) I read through the hyper book tutorial and cut/pasted & ran the examples in two evenings. Then read the first few chapters of the book (through the FIFO example, IIRC) slowly and carefully over another couple evenings. At that point I couldn't keep myself from starting to work on the model, and so I stopped reading, and cut over to modeling.

Modeling was slower – but really fun, so I kept at it. I ended up reading more bits from the book, and referring back to both the book and the tutorial a lot through the process. I worked on it at least some on most evenings over the next week and a half (and a few of the evenings that's pretty much all I did from home until bed), plus a fair bit of time on the weekend as well. I had made the shift in thinking necessary to model effectively – I believe – from the initial reading, but was still pretty clumsy with the language and tools, and that took some time to get comfortable with.

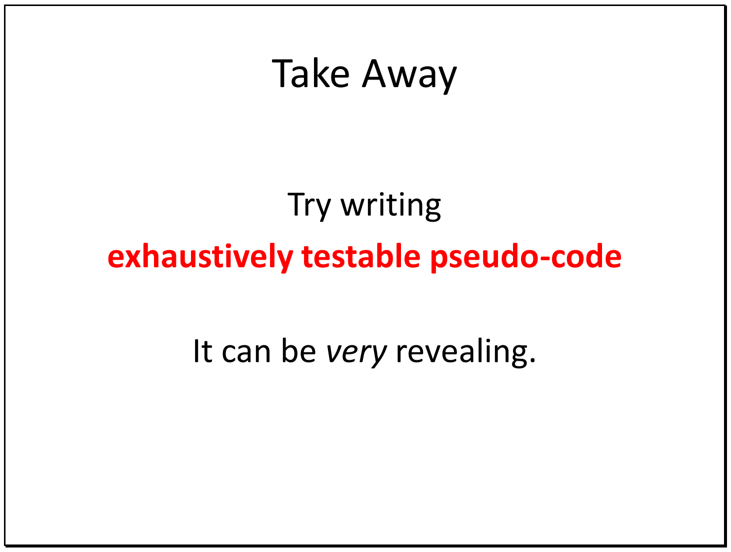
I'm absolutely sure that it helped a lot (in staying focused on learning it) that I had something very concrete that I wanted to have a model for that was part of my current dev work.

I also think it was very helpful that I remembered the basics of set theory from my discrete math course in college, and was familiar with functional programming. I think these two thing made the learning curve less steep for me than it would be for someone without those"



Getting started with Alloy

- IDE: Alloy Analyzer
- Tutorials: here and here.
- Jackson’s book: Book
- Community



Take Away

Try writing

exhaustively testable pseudo-code

It can be *very* revealing.