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
20s

According to citeseer, the 3rd most cited paper in computer science

http://citeseerx.ist.psu.edu/stats/articles;jsessionid=9B4212E42789A894069928BDD69878AC

Won sigcomm 2011 “test of time award”

http://www.sigcomm.org/awards/test-of-time-paper-award

Scrutinized far more than most designs.
Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance.

Chord is a dynamic ring of nodes, with an O(log N) overlay network. Can be used to build distributed hash-table key-value stores, which became the mainstay of the NoSQL movement.

The paper emphasizes simplicity and correctness. Informal proofs for Chord are in MIT TR 819.
20s
Not even listed by citeseer
Might have got more attention if it was called “How to quickly find major defects in your system design”

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

```alloy
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:>
}
```

“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 instances of all of the variables that satisfy the constraint (plus some ‘interesting condition’), or help understand why the constraint cannot be satisfied.
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.
A particular failure while a node is trying to join means that the node doesn’t actually join.
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
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.”
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”
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”.
Which pseudo-code language?

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)
Alpha SMP cache-coherency

Intel:
http://research.microsoft.com/pubs/64638/high-level.pdf

Security
http://profsandhu.com/it862/it862s05/ucon-tla1.pdf
A realistic example

Serializable Isolation for Snapshot Databases

Michael J. Cahill  
njpc@it.usyd.edu.au

Uwe Roehm  
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 interfering transactions that individually maintain consistency. Until now, Now used for 'SERIALIZABLE' isolation in PostgreSQL v9.1

They had to adapt the design: http://wiki.postgresql.org/wiki/Serializable

1. INTRODUCTION

Serializability is an important property when transactions execute because it ensures that logically consistent outcomes are maintained even if base constraints are not explicitly declared in the DML. If a DBMS enforces that all executions are serializable, then developers do not need to worry that inconsistent outcomes may occur.

Best paper award at SIGMOD 2008

Practical, realistic: <CLICK>

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
Pseudo-code from paper

modified read(T, x):
get lock(key=x, owner=T, mode=SREAD)
if there is a WRITE lock(wl) on x
    set wl.owner.isConflict = true
    set T.conflict = 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.conflict:
                abort(T)
        return UNSAFE_ERROR
    set xNew.creator.conflict = true
    set T.conflict = true

Figure 6: modified read(T, x)

Let’s make this exhaustively testable!
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}
  holdingLocks = ...
  ...
  waitingForLocks' = {}
  holdingLocks' = {key}
  ```

Counter-intuitive idea: the best way to model a system is a **single global state** with atomic transitions.


**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, pseudo-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
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.
Usually several actions are enabled in each state

Even a single action can lead to several next-states

Non-determinism. We want to exhaustively test ALL possible consequences (for some small number of transactions and keys).
Example of an ‘enabling condition’

\[
\begin{align*}
\text{StartedAndCanDoPublicOperation}(\text{txn}) &= \ \\
&= \left( \text{\textasteriskcentered Started and not yet finalized \textasteriskcentered} \right) \\
&\land \ \text{ txn } \in \text{ActiveTxns}(\text{history}) \\
&\quad \left( \text{\textasteriskcentered If \textasteriskcentered \text{txn} \text{ is waiting for a lock, then it cannot currently choose to commit, abort, read or write.} \textasteriskcentered} \right) \\
&\land \ \text{waitingForXLock[txn]} = \text{NOLock}
\end{align*}
\]

This is the enabling condition for all of:
- Commit(\text{txn})
- Read(\text{txn}, \text{key})
- Write(\text{txn}, \text{key})
- ChooseToAbort(\text{txn})
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] = {}]
```
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 than 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).
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”.
An interesting execution

A Read-Only Transaction Anomaly Under Snapshot Isolation
By Alan Fekete, Elizabeth O’Neill, and Patrick O’Neill
fekete@cs.umd.edu, {eoneill, pouriel}@cs.umd.edu
Research of all authors was supported by NSF Grant 07-11776.

Abstract. Snapshot Isolation (SI), a multi-version concurrency control algorithm introduced in [Bergman99] and later implemented by Oracle, is a transactional read-only transaction. However, it avoids concurrency errors, and it never delays read-only transactions. However, it is called its transactional lifetime. We say two transactions T<sub>i</sub> and T<sub>j</sub> are coincident if their transactional... preamble to create first version of each key (assumed by example in paper)

- preamble to create first version of each key (assumed by example in paper)
- preamble to create first version of each key (assumed by example in paper)

... the interesting part

- the interesting part

The example from the paper:

R<sub>2</sub>(X<sub>0</sub>,0) R<sub>2</sub>(Y<sub>0</sub>,0) R<sub>1</sub>(Y<sub>0</sub>,0) W<sub>1</sub>(Y<sub>1</sub>,20) C<sub>1</sub> R<sub>3</sub>(X<sub>0</sub>,0) R<sub>3</sub>(Y<sub>1</sub>,20) C<sub>3</sub> W<sub>2</sub>(X<sub>2</sub>,-11) C<sub>2</sub>

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

R<sub>1</sub>(X<sub>0</sub>) W<sub>1</sub>(Y<sub>1</sub>) W<sub>2</sub>(X<sub>2</sub>) C<sub>2</sub> B<sub>3</sub> C<sub>1</sub> R<sub>3</sub>(X<sub>2</sub>) R<sub>3</sub>(Y<sub>0</sub>) C<sub>3</sub>

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 = \{K<sub>0</sub>, K<sub>1</sub>\}, TxnId \{T<sub>0</sub>, T<sub>1</sub>, T<sub>2</sub>, T<sub>3</sub>\} with both configured as symmetry-sets of untyped model values)

... preamble to create first version of each key (assumed by example in paper)

- preamble to create first version of each key (assumed by example in paper)
- preamble to create first version of each key (assumed by example in paper)

... the interesting part

- the interesting part

\begin{verbatim}
[op -> "begin", txnid -> T<sub>0</sub>],
[op -> "write", txnid -> T<sub>0</sub>, key -> K<sub>0</sub>],
[op -> "write", txnid -> T<sub>0</sub>, key -> K<sub>1</sub>],
[op -> "commit", txnid -> T<sub>0</sub>],

[op -> "begin", txnid -> T<sub>1</sub>],
[op -> "read", txnid -> T<sub>1</sub>, key -> K<sub>0</sub>, ver -> T<sub>0</sub>],
[op -> "write", txnid -> T<sub>1</sub>, key -> K<sub>1</sub>],
[op -> "begin", txnid -> T<sub>2</sub>],
[op -> "write", txnid -> T<sub>2</sub>, key -> K<sub>0</sub>],
[op -> "commit", txnid -> T<sub>2</sub>],
[op -> "begin", txnid -> T<sub>3</sub>],
\end{verbatim}
[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:

\[ ~ \text{ReadOnlyAnomaly}(\text{history}) \]

with:

\[
\text{ReadOnlyAnomaly}(h) ==
\begin{align*}
& (\text{current history is not serializable} \ldots) \\
& \quad \Downarrow \quad \neg \text{CahillSerializable}(h) \\
& \quad (\text{... and there is a transaction that does some reads and zero writes,} \\
& \quad \quad \text{and, when that transaction is entirely removed} \\
& \quad \quad \text{from the history, the resulting history is serializable.}) \\
& \quad \Downarrow \quad \exists \txn \in \text{TxnId}:
\end{align*}
\]

\[
\begin{align*}
& \quad \text{LET keysReadWritten == KeysReadAndWrittenByTxn}(h, \\
& \quad \quad \txn) \\
& \quad \quad \quad \text{IN} \\
& \quad \quad \quad \quad \Downarrow \quad \text{Cardinality}(\text{keysReadWritten}[1]) > 0 \\
& \quad \quad \quad \quad \Downarrow \quad \text{Cardinality}(\text{keysReadWritten}[2]) = 0 \\
& \quad \quad \quad \quad \Downarrow \quad \text{CahillSerializable}(\text{HistoryWithoutTxn}(h, \txn))
\end{align*}
\]
TLC checking Cahill’s algorithm
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.