Notes for 6.826 lecture 2—The Amazon paper

The paper is

Newcombe et al, How Amazon Web Services Uses Formal Methods, *Comm. ACM* **58,** 4 (April 2015), pp 66-73.

(The paper on the course web page is an informal version dated 29 September 2014, but it appears to have the same content as the published version).

The paper refers to the TLA home page. Its current link is http://lamport.azureweb-sites.net/tla/tla.html

Talking points

Amazon is about *design*, not correctness. Insight vs. model check vs. proof

The best way to model a system is a single global state with atomic transitions.
A spec is a set of traces. Internal vs. external
Traces best represented by transitions of a state machine
Safety and liveness
Non-determinism >= concurrency

TLA: Two-state predicate to describe transitions:
- $x := x + 1$ becomes $x' = x + 1$. Actually $s' = s[x' := x + 1]$ (often written $s' = s[x + 1/x]$
- A disjunct for each transition. If more than one is enabled, non-determinism.
- Invariant

Stages:
- Write down the state.
- Write down (name and specify) the interface (possible state transitions).

Correctness of code by abstraction relation and simulation
- "If your system doesn't have a spec it cannot be wrong, it can only be surprising"
- Traces of code ⊆ traces of spec

How do you know the spec is correct? Keep it simple.
- Also, show that desired properties hold.

Stages:
- Write down the state.
- Write down (name and specify) the interface (possible state transitions).
- Write the abstraction relation from the code.
- Do a simulation proof.

Model checking. Amazon: We have found that model checking dramatically beats proof.

Why? It's automatic, and it finds most bugs, which show up on small examples.

This is very different from doing formal correctness proofs in Coq. It's less conclusive, but *much* cheaper.

Examples for specs:

Notation

Set comprehension is an important tool to avoid loops and recursion: $\{x{:}\,T \mid p(x)\}$ is the set of values x of type T such that $p(x)$ is true. Often it's a subset of another set s, written $\{x \in s \mid p(x)\}$. If the set is defined by some expression e involving x, write it $\{x{:}\,T \mid p(x) \mid e(x)\}$. To list the elements of a set explicitly: $\{a, b, c\}$.

The state is declared with **var**, actions with $name(args){:}\,resultType$. In an action **var** introduces a variable with an arbitrary value. **var** $x{:}\,T$, **var** $x \in s$ and **var** $x \in s \mid p(x)$ give constraints on x just like the ones for set comprehensions.

Useful vocabulary for specs: A sequence **seq** T is a function from a prefix of Nat to T. We write $i\,..\,j$ for the sequence $0, 1, \ldots, j$ and $f \circ g$ for the composition of f and g: first apply f, then apply g. Precisely

$$(f \circ g)(x) = g(f(x))$$

If f is a function, $f.dom$ and $f.rng$ are its domain and range, so if q is the sequence a, b, c then $q.dom = \{0, 1, 2\}$ and $q.rng = \{a, b, c\}$. $f[x \coloneqq y]$ has the same value as f at every argument except x, where its value is y.

If f and g are functions, $f + g$ is f overlayed by g, that is, g provides the value unless it's $None$, in which case f provides it: $f + g = \big(\textbf{fun}\ x \Longrightarrow \textbf{if}\ g(x) \neq None\ \textbf{then}\ g(x)\ \textbf{else}\ f(x)\big)$

If q is a sequence or set, $map(f, q)$ is $q \circ f$, the sequence or set obtained by applying f to each element of q, and $reduce(f, q)$ is the elements of q stuck together with f: $f(\ldots, f(f(q_0, q_1), q_2), \ldots, q_n)$, which is easier to read with f as a binary operator: $q_0 \boxed{f}\, q_1 \,\boxed{f}\, q_2 \boxed{f} \ldots \,\boxed{f}\, q_n$. Usually f is associative. Example: if q is the sequence of strings {"a","b","c"} then $reduce(++, q)$ is "abc". A prettier way to write it is $(++ : q)$, read "++ over q".

It's often clearer to write $f(x)$ as $x.f$ to cut down on nested parentheses and because $x.f.g$ reads easily as "start with x, apply f, then apply g.

If there are several terms in an expression with the same operator between them, such as

$$a \wedge b \wedge c$$

it's often easier to read if each term is on a separate line with an extra ∧ prefixed:

$$\begin{array}{l} \wedge\, a \\ \wedge\, b \\ \wedge\, c \end{array}$$

A powerful tool to avoid recursion is transitive closure. It is tricky, so it's fortunate that it isn't needed very often, but it is needed for symbolic links in directories. If r is a relation **set** (T, T) (which we often think of as a graph) then $r.closure$ is a relation that contains (t_1, t_2) iff there's a sequence of T's starting with t_1 and ending with t_2 such that r relates every pair of neighbors:

$$r.closure = \big\{t_1, t_2 \mid \exists q{:}\,\textbf{seq}\ T \mid q.head = t_1 \wedge q.last = t_2 \wedge \big(\forall i \in q.tail.dom, r(q_i, q_{i+1})\big)\big\}$$

Operationally, you can compute a finite closure rc iteratively one step at a time, starting with $rc = r$ and adding $\{t_1, t_2, t_3 \mid (t_1, t_2) \in rc \wedge (t_2, t_3) \in rc \mid (t_1, t_3)\}$ at each step until rc stops changing.

Often we want to relate just the elements of some set s and the "last" values that are related to them.

$r.fringe(s{:}\,\textbf{set}\ T) = \{t_1, t_2 \mid t_1 \in s \wedge r(t_1, t_2) \wedge \forall\, t{:}\,T \mid \neg r(t_2, t)\}$

If r is a function f, $f.closure.fringe$ is the fixed point, the result of applying f repeatedly until the result doesn't change.

Reliable two-party channel

var ch : **seq** Msg channel

$send(msg)$ $= ch := ch + + \{msg\}$
$receive(addr)$ $= msg = ch.head \rightarrow ch := ch.tail;$ **ret** msg
The arrow → is implication.

With crashes of sender or receiver some messages can be lost. This is non-deterministic, since we don't say exactly which messages are lost (perhaps none).

$crash$ $= \textbf{var}\ keep \subseteq ch.dom; ch := keep.sort \circ ch$

Async messaging

The channel can drop (lose) a message any time, and can deliver the same message any number of times.

type Msg $= (from, to: Addr, body: String)$ message (packet)
var ch : **set** Msg channel

$send(msg)$ $= ch := ch \cup \{msg\}$
$receive(addr)$ $= \textbf{var}\ msg \in ch | msg.to = addr;$ **ret** msg doesn't drop from *ch*

The $drop$ action is internal, and non-deterministic.
$drop(msg)$ $= \textbf{var}\ msg \in ch;\ ch := ch - \{msg\}$ can drop any time

Files

We describe just one file, and write just one byte at a time, to avoid fussy details about ranges.

type $File$ $=$ **seq** $Byte$
var f : $File$
$read(i)$ $=$ **ret** $f(i)$
$write(i, b)$ $= f := f[i := b]$ f changes just at i

Notation

$choose$ selects one element from a set; which one is arbitrary but deterministic, that is, the same for the same set. If $s$is empty, $s.choose = None$

$asFun$ turns a set of pairs into a function, which returns $None$ for any argument that isn't $from$ in some pair, otherwise the to value in some such pair.

type $Pair$ $= (from: T1, to: T2)$
$asFun(ps: \textbf{set}\ Pair) = (\textbf{fun}\ t \Longrightarrow \{p \in ps \mid p.from = t\}.choose)$

After a crash, any subset of the writes done since a $sync$ survive. So the following spec is highly non-deterministic:

type $NewVal$	$= (from\colon Nat, to\colon Byte)$	
var f	$:\ File$	copied from the basic spec
$fStable$	$:\ File$	as of the last *sync*
new	$:$ **set** $NewVal$	writes not yet synced
$read(i)$	$=$ **ret** $f(i)$	copied from the basic spec
$write(i, b)$	$= f := f[i := b]; new := new \cup \{(i, b)\}$	new f copied, write added to *new*
$sync()$	$= fStable := f;\ new := \{\}$	all the writes are in $fStable$
$crash()$	$=$ **var** $keep \subseteq new;\ f := f + keep.asFun$	keep some of the writes

Another way to say this is to keep track of all the possible post-crash files. This seems extravagant, since it makes lots of copies of the whole file, but there's no notion of cost for a spec. You can see that it's clearer and shorter.

var f	$:\ File$	copied from the basic spec
$fPost$	$:$ **set** $File$	all possible post-crash files
$write(i, b)$	$= f := f[i := b];$	
	$fPost := fPost \cup \{fp \in fPost \mid fp[i := b]\}$	
$sync()$	$= fPost := \{f\}$	now only the current file
$crash()$	$=$ **var** $newF \in fPost, f := newF$	choose some possible file

The choice of post-crash file is non-deterministic, so we can't say $fPost.choose$, because although that's an arbitrary value from $fPost$ it's a deterministic choice—$choose$ always makes the same choice from the same set, because only actions can be nondeterministic, not expressions.

The spec for non-atomic writes is very similar. There's a $done$ action which is much like $sync$ for a set of single-byte writes (think of a multi-byte write from one thread) and a variable $fInFlight$ which gives all the file states that a concurrent thread could observe. It's more complicated to spec several concurrent writes, because you have to keep track of which states survive a $done$ and which are intermediate states that are no longer visible.

All that was spec, mostly for what happens after a crash. Now for some code, modeled on the standard Unix data structures, but abstracting the details of packing an inode onto the disk:

type $Block$	$=$ **seq** $Byte$	disk block (sector)
$Addr$	$= Nat$	disk address
IN	$= (data\colon$ **seq** $Addr, length\colon Nat)$	inode; ignore how it fits on disk
INN	$= Nat$	inode number
Fc	$= INN$	file in code is just an inode no
var fs	$:\ Fc \to IN$	just the inodes
$disk$	$:\ Addr \to Block$	

Abstraction function. The abstract and concrete state of a file are marked like **this**. You concatenate the contents of the disk blocks with $reduce$ and take the first $fc.length$ bytes.

abs f $= reduce(concat, map(disk, fs(fc).data))(0\,..\,fs(fc).length - 1)$

Directories

The standard way to describe directories is to explain how to navigate from one directory to another in looking up a path name. Instead, this spec is in terms of the graph of directories. There's no recursion.

type $Node$	$= File \mid Dir$	
$Link$	$= (from, to: Node, name: String)$	node to node link
G	$=$ **set** $Link$	a whole directory graph
$Path$	$=$ **seq** $Link$	connected: $p(i).to = p(i+1).from$
PN	$=$ **seq** $String$	Path Name

Computed values of g. These make sense for a graph in general, except for the parts about names.

$nodes(g) = map(from, g).asSet \cup map(to, g).asSet$

A path is a connected sequence of links in g. If there are cycles this set is infinite.

$paths(g) = \{p: Path \mid p.asSet \subseteq g \wedge (\forall\, i \in p.tail.dom, p(i).to = p(i+1).from)\}$

An acyclic path doesn't visit the same node twice, so this set is finite if g is. We don't use this here.

$aPaths(g) = \{p \in paths(g) \mid \big((p \circ to).set \cup p.head.from\big).size = p.size + 1$

The summary of a path is a link:

$asLink(p) = (from := p.head.from, to := p.last.to, name := map(name, p).asString)$

To give it a $name$ we need a way to turn a path name into a string and vice versa:

$asString(pn) = reduce\big(++, (\textbf{fun}\ s \Rightarrow s + + \text{"/"})\big).removeLast$

$asPN(s) = \{pn \mid pn.asString = s\}.choose$ the set should have just one element

type FS	$= (g: G, root: Dir)$	File System
var tFS	$: FS$	the tree, without "." and ".."
$files(fs)$	$= \{n \in fs.g.nodes \mid n.type = File\}$	
$pathsTo(fs, n)$	$= \{p \in fs.g.paths \mid p.head.from = fs.root \wedge p.last.to = n\}$	

These are the properties of the file system tree graph $tFS.g$. They are invariants that follow from all the transitions. Here g and $root$ are short for $tFS.g$ and $tFS.root$: **let** $g = tFS.g, root = tFS.root$ **in**

All the links from a node have different names:

$$\forall\, l_1, l_2 \in g, (l_1.from = l_2.from) \Longrightarrow l_1.name \neq l_2.name$$

Only directories have outgoing links:

$$\forall\, l \in g, l.from.type = Dir$$

Only files have more than one incoming link:

$$\forall\, n \in g.nodes - g.files, \{l \in g \mid l.to = n\}.size \leq 1$$

The root has no incoming links:

$$\forall\, l \in g, l.to \neq root$$

Every node is reachable from the root via some path, which is unique except for file nodes:

$$\forall\, n \in g.nodes - \{root\}, \textbf{let}\ p = pathsTo(n)\ \textbf{in}\ p \neq \{\} \wedge (n.type \neq File \rightarrow p.size = 1)$$

It follows that the graph is a tree except for multiple paths to files.

Dot names. To describe the real file system we have to add "." and "..". Every non-file gets a "." *Link* to itself, and each one except the root gets a ".." link to its unique parent.

$addDots(fs)$ $= \textbf{let}\ ds = \{n \in fs.g.nodes \mid n.type = Dir\}\ \textbf{in}$
$fs[g := fs.g \cup \{n \in ds \mid (n, n, ".")\}$
$\cup \{n \in ds - \{fs.root\} \mid (n, fs.pathsTo(n).last.from, "..")\}$
var dFS $= addDots(tFS)$

Looking up file names. A pn that starts with the empty string corresponds to a file name that starts with "/", and needs special treatment (first line). Then we look for a path that matches pn, returning the node at the end of the path if there is one and $None$ otherwise. This doesn't depend on any FS invariants except that names on outgoing links are unique, which ensures that ps is a singleton or empty.

$lookup(fs, pn: PN, cd: Dir): \textbf{option}\ Node$
$= \textbf{let}\ d = \textbf{if}\ pn.head = "" \ \textbf{then}\ fs.root\ \textbf{else}\ cd\ \textbf{in}$ — empty head → start with $root$, not cd
$\textbf{let}\ ps = \{p \in paths(fs.g) \mid$
$p.head.from = d \wedge map(name, p) = pn\ \textbf{in}$
$\textbf{if}\ ps \neq \{\}\ \textbf{then}\ ps.choose.last.to\ \textbf{else}\ None$ — if there is a path return its last node

Symbolic links are also in the real file system, and they are trickier. Since $lookup$ fails if it encounters an $sLink$, the idea is to resolve all the $sLink$'s whose targets can be resolved without traversing any $sLink$ that hasn't already been resolved. When an $sLink$ is resolved, all the links to it are replaced by links to the target. Since there are no links from it, it's no longer part of g.

What about the root? If it's an $sLink$, there's no link to it, so it doesn't appear in old or new below. Its target becomes the new root. [[I wonder whether this is right?]]

type $Node$ $= File \mid Dir \mid SLink$
$SLink$ $= String$
$resolve(fs): FS = \textbf{let}\ g = fs.g, root = fs.root\ \textbf{in}$ — short names for g and $root$
$\textbf{let}\ tgt = \{n \in g.nodes \mid n.type = SLink$ — for each *sLink*
$\mid (from := n, to := fs.lookup(n.asPN, n))$ — look up the target name in fs
$\}.asFun\ \textbf{in}$ — and make $sLink \rightarrow target$ if it works
$\textbf{let}\ old = \{l \in g \mid l.to.tgt \neq None\}\ \textbf{in}$ — the old links to resolved *sLink*s
$\textbf{let}\ new = \{l \in old \mid l[to := l.to.tgt]\}\ \textbf{in}$ — the new links to their targets
$fs[g := (g - old) \cup new,$ — replace old links with resolved ones
$root := \textbf{let}\ r = root.tgt\ \textbf{in}$ — and if the root was a resolved *sLink*
$\textbf{if}\ r \neq None\ \textbf{then}\ r\ \textbf{else}\ root\ ]$ — make its target the new root

Now we apply the transitive closure of the $resolve$ relation (which is actually a function) to dFS get the graph in which we actually want to look up file names:

var sFS $= resolve.asRel.closure.fringe.asFun(dFS)$

This terminates, because it resolves at least one $sLink$ at each step in computing the closure, and is done if it can't do so. If there's a cycle of target names, for example if the target of `/a/b` is `/a/b/c`, then the $sLink$s that are part of such a cycle won't get resolved by $resolve$.

Once the links are resolved many of the invariants on tFS no longer hold: there can be cyclic paths, multiple paths to directories, etc. But $lookup$ doesn't depend on these invariants; it just follows pn.

Of course the code that handles $sLink$s in Unix doesn't construct a sFS in which all the links are resolved. Instead it follows the path given by pn, and when it comes to an $sLink$ named sl with target tgt, it starts looking up $tgt +\!+ pn'$, where $pn = sl +\!+ pn'$. If there's a cycle this won't terminate, so there needs to be an explicit check for revisiting the same $sLink$ or a bound on the number followed.

Excerpts from the Amazon paper

Some of the more subtle, dangerous bugs turn out to be errors in design; the code faithfully implements the intended design, but the design fails to correctly handle a particular ‘rare’ scenario.

All models are wrong, some are useful.

We needed to be able to capture the essence of a design in a few hundred lines of precise description. We found what we were looking for in TLA+. A TLA+ specification describes the set of all possible legal behaviors (execution traces) of a system.

The TLC model checker [5], a tool which takes a TLA+ specification and exhaustively checks the desired correctness properties across all of the possible execution traces.

Our experience with TLA+ has shown that perception to be quite wrong. So far we have used TLA+ on 10 large complex real-world systems. In every case TLA+ has added significant value, either finding subtle bugs that we are sure we would not have found by other means, or giving us enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness.

We specify the effects of each of those possible events; e.g. network errors and repairs, disk errors, process crashes and restarts, data center failures and repairs, and actions by human operators.

All production services at Amazon are under constant development: new features, increases in scale, removing bottlenecks. Many of these changes are complex, and they must be made to the running system with no downtime.

Spec is also an excellent form of documentation, very important as our systems have unbounded lifetime.

Exhaustively testable pseudo-code.

We have adopted the practice of first writing a conventional prose design document, then incrementally refining parts of it into PlusCal or TLA+. Often this gives important insights without ever going as far as a full specification or model checking.

“How do we know that the executable code correctly implements the verified design?” The answer is that we don’t. Despite this, formal methods help in multiple ways:

- Get the design right.
- Gain a better understanding of the design.
- Find strong invariants

The designer must ensure that the model captures the significant aspects of the real system. Achieving this is a difficult skill, the acquisition of which requires thoughtful practice. Also, we were solely concerned with obtaining practical benefits in our particular problem domain.

What is formal specification is not good for? Surprising ‘sustained emergent performance degradation’.