Notes 8

7.1 Why do Proofs?

- Convince yourself that your reasoning is correct – that you haven't made mistakes.
- Convince others.
- A good proof has considerable structure, like a good program. Much of it can be reused for other related results (like a well-written program).
- A proof pins down *why* something is true, so you can know whether it will stay true when things change.
- A failure to prove something often reveals the reason it is false.

Program verification: proof that a program does the right thing.

- Pentium
- Airbus A300 (early version of control software had a bug that caused plane to head straight down if a certain configuration was reached)
- Therac 25

In programming, the current state of the art is that people do not actually prove correctness of running programs. There has been a lot of research directed toward making this possible, but it hasn't yet gotten far enough to be practical. The problem is that the concepts that have to be reasoned about seem too low-level, complicated in their details.

In contrast, people do know how to model programs/systems abstractly, and prove correctness of the abstract versions. E.g., the *algorithm* used in a communication protocol can be described formally, and its correctness proved. But there is still a gap between what is verified and the actual code that runs on real systems – programmers implement algorithms in a rather ad hoc manner, and errors can creep in there. There isn't much available now to help programmers bridge this gap.

The high-level algorithm proofs are usually done as sketches. Some are very complicated, and some errors have been found in published proofs. There is also considerable research in actually doing complete algorithm proofs axiomatically, and having a computer tool (an "automatic theorem-prover") help in checking or even filling in the low-level steps. This sort of work is becoming practical.

Whether done by hand or with computer assistance, algorithm proofs are of great help in getting your programs right (and understanding why). This will probably be done much more in the future. It will become an important part of quality assurance for programs. It takes a lot of work to do it, and you have to understand formal logic proofs before you do it.

Other examples:

- Kaashoek's Orca system, for which Alan Fekete and Nancy Lynch helped to verify the main algorithm (and found a significant logical error in the process of doing the proof).
- Current chip designs – companies like Intel, HP are now investing in verification people and computer tools.
- Communication protocols, AT&T.
- Practical verification efforts are also beginning for real-time process control systems, e.g., those systems that control nuclear power plants and airplanes. Still preliminary.

Program verification is a nice complement to testing – testing can often detect bugs quickly. Verification is harder, but it does help to find deeper logical mistakes. And when you're done, you know there are *no more* errors (which testing alone can never guarantee).

8 Distributed Algorithms

The technique of modeling programs as state machines in order to reason about their behavior (reachable states, invariants) extends to parallel programs as well as sequential.
Even to *distributed algorithms*, in which there are many programs running independently at different nodes of a network.

8.1 Leader Election

Token-based leader election, in a ring of processes.

Imagine a collection of processes numbered $1, \ldots, n$, arranged in a ring (mod n).
Want to elect exactly one as a leader.
Each starts with a *unique identifier*, say a natural number.
(Any combination of identifiers can appear in the ring, as long as there are no repeats.)

Each process sends its own identifier clockwise to its neighbor, pass around ring.
But: Each process only passes on identifiers greater than its own.
If a process receives its own identifer, it elects self leader.
(The idea is that only the max uid will succeed in getting all the way around the ring.)

Formally, state consists of:
For each i, $1 \leq i \leq n$:
$val(i)$, i's identifier (initially, all are unique)
$buffer(i, i+1 \bmod n)$, a finite sequence, initially containing just $val(i)$
(Communication buffer for messages from process i to process $i+1$.)
$leader(i)$, a Boolean, initially *false*

$move(i)$
Pre:
$buffer(i-1, i)$ is nonempty
Eff:
let $t = head(buffer(i-1, i))$
remove *head* of *buffer*
if $t > val(i)$ then add t to end of $buffer(i, i+1)$
if $t = val(i)$ then $leader(i) := true$

We would like to prove that exactly one $leader(i)$ flag gets set.
For now, we'll content ourselves with just showing that *at most* one flag is set.

Invt1: At most one $leader(i)$ is *true*.

Needs strengthening:

Invt2:
Loosely: "A uid in a buffer is greater than the uids of all processes that it has passed."
More formally: If $val(i)$ is in $buffer(j, j+1)$, then $val(i)$ is greater than all of $val(i+1)$, $val(i+2), \ldots, val(j)$.

Can prove by induction.
Key is that a node only lets the value pass if it's greater.

Showing some leader is eventually elected is a liveness property; however, showing it uses an auxiliary invariant saying that the maximum uid is always *somewhere* in the state.

(Conservation of max uid).

8.2 Alternating Bit Protocol

A common communication protocol.
Sending a sequence of messages from a set M, from process 1 to process 2.
Using underlying FIFO communication channels, can lose or duplicate messages.
Process 1 keeps sending first message tagged with 1.
When process 2 receives, accepts and outputs.
Acks (repeatedly) with tag 1.
When process 1 receives 1, starts sending next message, tagged with 0.
Process 2 ignores tag 1, until it sees first 0; then accepts and outputs, and then acks repeatedly with 0.
Process 1 now waits to receive 0, then goes back to tag 1.
Alternates.

The claim is that this causes process 2 to output a *prefix* of the sequence of process 1.
(To see that it gives all requires a liveness argument.)

States:
buf_1, a finite sequence of M elements, initially arbitrary,
$work\text{-}buf_1$, same, initially equal to buf_1
(This represents the remaining messages to be sent, whereas buf_1 represents all the messages ever submitted.)
$tag_1 \in \{0,1\}$, initially 1.

buf_2, a finite sequence of M elements, initially empty,
$tag_2 \in \{0,1\}$, initially 0.

$channel_{1,2}$, a finite sequence of elements of $M \times \{0,1\}$, initially empty,
$channel_{2,1}$, a finite sequence of elements of $\{0,1\}$, initially empty

$send(m,b)_{1,2}$
Precondition:
$m = head(work\text{-}buf_1)$
$b = tag_1$
Effect:
add (m,b) to end of $channel_{1,2}$,
any number ≥ 0 of times

$send(b)_{2,1}$
Precondition:
$b = tag_2$
Effect:
add b to end of $channel_{2,1}$,
any number ≥ 0 of times

$lose_{2,1}$
Precondition:
$(channel_{2,1})$ nonempty
Effect:
remove *head* of $channel_{2,1}$

$copy_{2,1}$
Precondition:
$(channel_{2,1})$ nonempty
Effect:
copy *head* of $channel_{2,1}$

$receive(m,b)_{1,2}$
Precondition:
$(m,b) = head(channel_{1,2})$
Effect:
remove *head* of $channel_{1,2}$
if $b \neq tag_2$ then
add m to buf_2
$tag_2 := tag_2 + 1 \bmod 2$

$receive(b)_{2,1}$
Precondition:
$b = head(channel_{2,1})$
Effect:
remove *head* of $channel_{2,1}$
if $b = tag_1$ then
remove head of $work\text{-}buf_1$ (if nonempty)
$tag_1 := tag_1 + 1 \bmod 2$

$lose_{1,2}$
Precondition:
$(channel_{1,2})$ nonempty
Effect:
remove *head* of $channel_{2,1}$

$copy_{1,2}$
Precondition:
$(channel_{1,2})$ nonempty
Effect:
copy *head* of $channel_{1,2}$

Goal invariant, Invt1:
buf_2 is a prefix of buf_1

Needs intermediate invariant:

Let *tag-sequence* be the sequence consisting of elements of $channel_{2,1}$,
followed by tag_2,
followed by the tag components of the elements of $channel_{1,2}$,
followed by tag_1.

Use sequence concatenation notation.

Invt2:
tag-sequence consists of one of the following:

1. All 0's.
2. All 1's.
3. A positive number of 0's followed by a positive number of 1's.
4. A positive number of 1's followed by a positive number of 0's.

What is being ruled out is more than one switch of tag value.

This is proved easily by induction (no extra hypotheses needed);

Also need:

Invt3:
If (m, tag_1) is in $channel_{1,2}$ then $m = head(work\text{-}buf_1)$.

This again follows by induction. Uses Invt2 in the proof.
(BASE: Obvious, since no message is in the channel initially.
INDUCTIVE:
If add new message to channel, obvious.
Only interesting case is if tag_1 changes and there is something else already in $channel_{1,2}$ with the same tag.
But this can't happen, by Invt2 applied before the step – since incoming tag $= tag_1$, all the tags in the state must be the same.)

Finally, need:
Invt4:
1. If $tag_1 \neq tag_2$ then:
$buf_2 \cdot work\text{-}buf_1 = buf_1$.
2. If $tag_1 = tag_2$ then
$work\text{-}buf_1$ and buf_2 are nonempty,
$head(work\text{-}buf_1) = last(buf_2)$, and
$buf_2 \cdot tail(work\text{-}buf_1) = buf_1$.

$tail =$ all but head.

Invt4, part 2, easily implies the goal Invt1.
Also implies that $work\text{-}buf_2$ is always nonempty when the $receive_1$ occurs for equal tags.
(So parenthetical check is always true.)
Prove Invt4 by induction.
BASE:
tags unequal, $work\text{-}buf_1 = buf_1$ and buf_2 empty.
Suffices.

INDUCTIVE STEP:
When send, don't change tags or buffers.
So can't change the truth of the invariants.
So only consider receives.

$receive(m, b)_{1,2}$:
If $b = tag_2$, nothing happens, so preserved.
If $b \neq tag_2$, arriving message is same as first on $work\text{-}buf_1$, by Invt3.
Change tag to make equal, replicate $first(work\text{-}buf_1)$ at end of buf_2.
Since concatenation previously held (by inductive hypothesis), now the modified concat does.

$receive(b)_{2,1}$
Similar argument.
If $b \neq tag_1$, nothing happens, preserved.
If $b = tag_1$, change tag to make unequal, remove first element of work-buf.
Since modified concat previously held (by inductive hypothesis), now ordinary concat does.