Massachusetts Institute of Technology | Lecture ??
6.042J/18.062J: Mathematics for Computer Science | October 10, 1996
Professors Nancy Lynch and Tom Leighton

Notes 11

In this lecture, we finish state machines.
An important discrete structure used for modeling both algorithms and physical machines.
(Both software and hardware.)
Review definitions and features highlighted by examples of last time.
Introduce invariants and how to prove them by induction.
Then do more examples.

1 Review

A *state machine* is a triple (Q, Q_0, δ), where:

1. Q is a nonempty (finite or infinite) set of *states*.
2. $Q_0 \subseteq Q$ is a nonempty subset of the states known as the *start states*.
3. $\delta \subseteq Q \times Q$ is a binary relation over Q.

Like a *binary relation*; formally, the only difference is the specially distinguished start states.

The point of a state machine is that it *executes*, producing *executions*.
Starts from any start state, and moves along, following transitions.
In this way, it models a computational process.

An *execution fragment* of a state machine is a finite or infinite sequence $q_0, q_1, \ldots$ such that $(q_i, q_{i+1}) \in \delta$ for every $i \geq 0$.

An *execution* is an execution fragment in which the initial state q_0 is an element of Q_0.

A state of Q is said to be *reachable* if it is the final state of some finite execution.

We did several examples.
Ex. 1: A simple counter. This was an example of a *deterministic* state machine. Just one start, 0, and one next place to go, by incrementing. Formally, $|Q_0| = 1$, and for every $q \in Q$, there is at most one q' such that $(q, q') \in \delta$.

A state machine that isn't deterministic is called *nondeterministic*.

Then did some hopscotch examples. These were all *nondeterministic* state machines, because they had options of going east or west, (or north or south in the 2-dimensional case).

Ex. 2: Linear hopscotch
Allowed steps of 12 or 18 in either direction, + or −.

Then we introduced the idea of explicitly representing the *structure* of the state machine's state by using *state variables*.
This can almost always be done in practice.

Ex. 3: 2-dimensional hopscotch
With 12 and 18 in EW and 9 and 12 in NS direction.
Q: Consists of values for two *variables*:
east-west $\in \mathbb{Z}$, initially 0
north-south $\in \mathbb{Z}$, initially 0

Q is the Cartesian product, $\mathbb{Z} \times \mathbb{Z}$.
Q_0 consists of the single pair, $(0, 0)$.

δ:
Transitions of the following forms:

move-east(12)
Effect:
east-west := *east-west* + 12

move-east(18)
Effect:
east-west := *east-west* + 18

move-west(12)
Effect:
east-west := *east-west* − 12

move-west(18)
Effect:
east-west := *east-west* − 18

move-north(9)

Effect:
$north\text{-}south := north_south + 9$

Etc.

Then showed how sometimes it's convenient to give a predicate restricting when the code fragment is allowed to be applied:

Ex. 4:
2-dim hopscotch on only northeast quadrant:

The same as above, only now the two state components *east-west* and *north-south* take on nonnegative values only (values in $\mathbb{N}$).

E.g.:
move-west(12)
Precondition:
$east\text{-}west \geq 12$
Effect:
$east\text{-}west := east\text{-}west - 12$

Finally, introduced important idea of an *invariant*, the main notion used in practical methods for formal reasoning about the correctness of algorithms/machines represented as state machines.

Def.: An *invariant* of a state machine M is any predicate that is true of all the reachable states of M.

Observed that the general 2-dimensional hopscotch has the invariant that *east-west* is a multiple of 6,
and *north-south* is a multiple of 3.

To prove these are invariants, we carried out an inductive proof on the length of an execution of the 2-dimensional hopscotch state machine.

Note this is essentially what we did earlier, in the inductive proof lectures, 2 and 3, to reason about a simple bitwise multiplication algorithm.

Today, we will do more examples.
Illustrate how state machines are used to represent typical computational processes,

and how interesting properties of those processes can be proved as invariants, by induction.

Examples will be chosen from number theory, posets, graphs, and distributed computing.

2 Number-Theoretic State Machines

Euclid's GCD algorithm, fast multiplication

2.1 Euclid's GCD Algorithm

Recall what it does:
Takes $x, y \in \mathbb{N}$, initially arbitrary, subject to: not both 0, $x \geq y$.
Returns $\gcd(x, y)$.

Q consists of state components:
$x, y \in \mathbb{N}$, initially arbitrary, subject to: not both 0, $x \geq y$
These are the values for which we want to find the GCD

Also need auxiliary components:
$x_1, y_1 \in \mathbb{N}$, the current values on which we're working
Initially, $x_1 = x$, $y_1 = y$

And a component to hold the final answer:
$answer \in \mathbb{N} \cup \{null\}$, initially $null$.

δ:

Two kinds of things happen: recursive calls, and final decisions.

recursive-call
Precondition:
$answer = null$
$y_1 \neq 0$
Effect:
$(x_1, y_1) := (y_1, x_1 \bmod y_1)$

final-decision
Precondition:
$answer = null$
$y_1 = 0$

Effect:
$answer := x_1$

Would like to prove:
If $answer \neq null$ then $answer = \gcd(x, y)$.
This says that the state machine, if it ever reaches an answer, reaches the correct answer (partial correctness).

Can't prove just this by induction – not strong enough.
Need to strengthen the inductive hypothesis (as usual), this time to say something about the case where $answer = null$.

Here, what works is a 2-part invariant:

1. If $answer \neq null$ then $answer = \gcd(x, y)$.

2. If $answer = null$ then $\gcd(x_1, y_1) = \gcd(x, y)$.

Now can prove this 2-part invariant by induction on number of steps in an execution.
BASE: 0 steps.
In initial state, $answer = null$.
Since $x_1 = x$ and $y_1 = y$, obviously $\gcd(x_1, y_1) = \gcd(x, y)$.

INDUCTIVE STEP: Assume true after n steps and show after $n + 1$ steps.
So consider and $n + 1$-step execution.
It arises from an n-step execution, by adding one step at the end.
Consider cases based on the type of action – *recursive-call* or *final-decision* – in that last step.

If it's *recursive-call*, then the precondition says that before the last step, $answer = null$; this step keeps it $null$.
So we are in the second case, and we must show that $\gcd(x_1, y_1) = \gcd(x, y)$ after the step.
We may assume this before the step, by inductive hypothesis.
That is, assume $\gcd(q.x_1, q.y_1) = \gcd(q.x, q.y)$.
Prove $\gcd(q'.x_1, q'.y_1) = \gcd(q'.x, q'.y)$.

This follows from:
(a) $q.x = q'.x$, $q.y = q'.y$ – these remain unchanged,
(b) the interesting fact, proved earlier in the course:
$\gcd(q.x_1, q.y_1) = \gcd(q.y_1, q.x_1 \bmod q.y_1)$,
and (c) the behavior of the code fragment, which ensures that
$q'.x_1 = q.y_1$ and $q'.y_1 = q.x_1 \bmod q.y_1$.

On the other hand, it the last step is *final-decision*, then the precondition says that $answer = null$ and $y_1 = 0$ before the last step;
after the last step, $answer \neq null$.
So we are in the first case, and we must show that $answer = \gcd(x, y)$.
We may assume the invariants before the step:
$\gcd(q.x_1, q.y_1) = \gcd(q.x, q.y)$.
We must prove $q'.answer = \gcd(q'.x, q'.y)$.

This follows from:
(a) $q.x = q'.x$, $q.y = q'.y$,
(b) the interesting fact that $\gcd(q.x_1, 0) = q.x_1$, and
(c) the behavior of the code fragment, which ensures that $q'.answer = q.x_1$.

This example has illustrated strengthening the hypothesis in order to prove it.
That strategy is very common in inductive proofs in general.
It is almost always used in proofs for state machines.

2.2 Fast Multiplication

"Russian peasant's multiplication"

The states are composed of variables, all in $\mathbb{N}$:
x, y, the values to be multiplied, initially arbitrary positive
a, the accumulator for the answer
r, s, temporaries

Initially, $r = x$, $s = y$, $a = 0$.

Transitions:

even
Pre:
s is even
$s > 0$
Eff:
$r := 2r$
$s := \frac{s}{2}$

odd
Pre:
s is odd
Eff:
$a := a + r$
$r := 2r$
$s := \frac{s-1}{2}$

The answer is the value in the accumulator a when $s = 0$.

This may not be completely obvious.
Example execution:

\$x\$	\$y\$	\$r\$	\$s\$	\$a\$
5	9	5	9	0
		10	4	5
		20	2	5
		40	1	5
		80	0	45

Proof: To see this is correct, would like to prove invariant:

Invariant 1: If $s = 0$ then $a = xy$.

Of course, need to strengthen in order to prove.
Turns out that the key is:

Invariant 2: $xy = rs + a$

Notice how this expresses the idea that explains why the algorithm works.
This is typical for invariants.

So, can prove Invt1 and Invt2 together by induction, as before.
Alternatively, and slightly better, can just prove Invt2 by itself, by induction, and then infer Invt1 as a corollary.

Proving Invt2 is straightforward by induction:

BASE:
True initially because $r = x$, $s = y$ and $a = 0$.

INDUCTIVE STEP:
Consider cases based on type of action, *even* or *odd*.

even:
Doubling r while halving s doesn't change the product rs,
and a doesn't change, so easy.

odd:
Assuming $(q.x)(q.y) = (q.r)(q.s) + q.a$.
Must show $(q'.x)(q'.y) = (q'.r)(q'.s) + q'.a$.

Now $q'.x = q.x$, $q'.y = q.y$, so LHS unchanged.
Also, $q'.r = 2(q.r)$, $q'.s = \frac{q.s-1}{2}$, and $q'.a = q.a + q.r$.
So $(q'.r)(q'.s) + q'.a = 2(q.r)(\frac{q.s-1}{2}) + q.a + q.r$.
Reorganizing terms yields $(q.r)(q.s) + q.a$, which suffices.

So Invt2 is proved.
For Invt1, don't even need an induction:

Already know that in all reachable states, $xy = rs + a$.
Need to show in all reachable states, if $s = 0$ then $a = xy$.

So let q be any reachable state.
We know that $(q.x)(q.y) = (q.r)(q.s) + (q.a)$.
If $q.s = 0$, we get $(q.x)(q.y) = q.a$.
As needed.

This has illustrated the idea of strengthening an invariant, then proving only a part of it by induction and getting the rest as a consequence (corollary).

This strategy extends:
Sometimes to prove an invariant must extend it to a long list.
Then useful to prove one or a batch first, by induction.
After that, we can assume their results in proving another batch by induction.
Etc.
Anywhere along the way, might get some as consequences, without using induction.

3 Posets

Algorithms can also manipulate other data structures besides just numbers.
Here we do algorithms that manipulate posets.
Comparison-based algorithms.

3.1 Bubble Sort

A comparison-based sorting algorithm.

Fix $n \geq 2$, length of sequence to sort.

State has:
a, a length n sequence of integers, write components $a(1), a(2), \ldots a(n)$.
Initially arbitrary.
i, j, pointers in range $[2, n+1]$, initially 2

i keeps track of which pass we're up to – pass corresponds to the initial position of the element that we're trying to insert in its correct position in the sequence.
j keeps track of our progress in trying to locate the proper position for the element we're currently working on.

Only one type of transition, though it has some cases:

compare
Precondition:
$i \leq n$
Effect:
if $a(j-1) \leq a(j)$ then $i, j := i+1$
else
interchange $a(j-1)$ and $a(j)$
if $j > 2$ then $j := j-1$ else $i, j := i+1$

Describe operation informally.
$a(j-1) \leq a(j)$ means have placed current element in correct position relative to ones that begin ahead of it.
Now go on to next pass, to place next element.

Otherwise, interchange. See if have reached top – if so go on to next element, if not, continue this pass.

Final invariant will be:
If $i = n + 1$ then a is sorted (in nondecreasing order).

To prove this by induction, need invariants representing partial progress:

1. $2 \leq j \leq i \leq n + 1$
   (Just gives range of index values.)
2. $i = n + 1$ if and only if $j = n + 1$.
   (Special case – if indices hit end we're done.)
3. $a(k) \leq a(l)$ for all k, l, $1 \leq k \leq l \leq \min(i, n)$, $k, l \neq j$.
   (That is, all elements up to and including i, except possibly the jth element, are in sorted order.)
4. $a(j) \leq a(k)$ for all k, $j + 1 \leq k \leq \min(i, n)$.
   (That is, the jth element is in the right place with respect to the elements in higher-numbered positions – up to and including ith position.)

Note that these statements represent the key reasons why this algorithm works.

Must argue these invariants are preserved.

Invt1 is trivial, prove by itself.
By induction.
Base: True in initial state, since $2 \leq 2 \leq 2 \leq n + 1$.
Inductive: Cases.
Could set $i, j := i + 1$, but only in case $i \leq n$. This implies $2 \leq j \leq i \leq n + 1$.
Could set $j := j - 1$, but only in case $j > 2$. This preserves.

Invt2 likewise trivial, by induction.

Invts 3 and 4 less trivial.
Prove them together, again by induction.
Argue in cases, based on which clause in the effect is followed.

Case 1:
The first test comes out positive, saying $a(j - 1) \leq a(j)$.
Informally, this means that the jth element is already in the right position relative to all others, so we have a total order for $a(1), \ldots, a(i)$.

As needed to start next phase with new i.
Formally, we need to show Invts 3 and 4 after the step.
These are both implied by the claim that $a(1) \leq a(2) \ldots a(q.i)$ after the step.
That is, the entire list through position $q.i$ (the old value of i) is now sorted.

Say why this implies both Invt3 and Invt4.
Invt3 excludes the last element $a(q'.i)$ (may not even exist, in case $i = n+1$) and Invt4 is vacuous.

So, show $q'.a(1) \leq q'.a(2) \ldots q'.a(q.i)$.

By inductive hypothesis, we know that $q.a(1) \leq q.a(2) \leq \ldots \leq q.a(q.j-1) \leq q.a(q.j+1) \leq \ldots \leq q.a(q.i)$.
Also, $q.a(j) \leq q.a(k)$ for all k, $q.j+1 \leq k \leq q.i$. By precondition, we know that $q.a(q.j-1) \leq q.a(q.j)$.

Draw Hasse diagrams, Figure 1.

This all implies that $q.a(1) \leq q.a(2) \leq \ldots \leq q.a(q.j-1) \leq q.a(q.j) \leq q.a(q.j+1) \leq \ldots \leq q.a(q.i)$.
That is,
$q.a(1) \leq q.a(2) \leq \ldots \leq q.a(q.i)$.
But the elements in the array are unchanged by the step.
So $q'.a(1) \leq q'.a(2) \leq \ldots \leq q'.a(q.i)$,
as needed.

Case 2:
The test comes out negative.
In this case, a single interchange occurs.
Informally, the new element is correctly ordered with respect to all later elements, after the exchange, which yields Invt4 in the new state.
And all other elements are still correctly sorted, which gives Invt3 in the new state.
There are then two subcases, based on whether we are done with this i.

The formal argument breaks up into these two cases.
Argument in same style as above.
LTTR to do carefully.
DO THIS TO MAKE SURE YOU UNDERSTAND WHAT'S GOING ON HERE.
Do in recitation.

E.g., need to use arguments like: $q.a(q.j) < q.a(q.j-1)$ implies $q'.a(q'.j-1) < q'.a(q.j)$

(after interchange).
Technical, must keep notation straight for before-after states.
Need distinction for all variables – indices too, since these sometimes change.

Once Invts 3 and 4 are proved, still need to show the final invt:
If $i = n + 1$ then a is sorted (in nondecreasing order).
Follows from Invt3, applied with $j = n + 1$ (using Invt2 to deduce that also $i = n + 1$).

4 Graph Algorithms

Algorithms can also manipulate graphs.

4.1 Acyclic Graph Maintenance

State is a directed graph G with no self-loops.
Represented by components V and E.
Initially, it is any acyclic graph (define).
Transition: Pick a node with only incoming edges.
Reverse all the edges simultaneously:

$reverse(v)$
Precondition:
E contains no edge of the form (v, w)
Effect:
let W $\{w \in V | (w, v) \in E\}$.
$E := E \cup \{(v, w) \mid w \in W\} - \{(w, v) \mid w \in W\}$

Invariant: Graph (V, E) is acyclic.
Prove by induction.
Inductive step:
Suppose graph is acyclic.
Claim one reversal can't introduce a cycle.
If it did, then one of the new edges inserted by the reversal would have to be involved in the cycle.
(Uses inductive hypothesis – no cycles in pre-state.) But each such edge starts at v, so the cycle would have to have an edge that ends with v.
But this can't be, since we are reversing *all* edges incoming to v.

4.2 Minimum Weight Paths ("Shortest Paths")

State contains a digraph G again, represented by components V and E.
Also v_0, a distinguished node in V.

Also, each edge (v, w) in E has a nonnegative real-valued *weight*, $weight(v, w)$.
(Could be different from $weight(w, v)$.)
Represented in matrix.

The goal is for each node to determine the total weight of the minimum-weight path from v_0 to itself in the digraph.
A.K.A. "shortest path", using strange mix of weight and length terminology.

Another state component:
Each node v has a path-weight estimate, $pweight(v)$, either a nonnegative real or ∞.
Initially, $pweight(v_0)$ is set to 0, and all the others are ∞.

There is only one type of transition, in which one of the vertices, v, improves (lowers) its path weight estimate based on the latest estimate of one of its neighbors, w:

$improve(w, v)$
Pre:
$(w, v) \in E$
$pweight(w) + weight(w, v) < pweight(v)$
Eff:
$pweight(v) := pweight(w) + weight(w, v)$

Very nondeterministic – this has lots of different executions, depending on which vertices improve based on which neighbors, in which order.
What happens?
First, at any time, for each vertex that has a finite estimate, the estimate corresponds correctly to some path from v_0 to itself.
Second, if all vertices keep trying to improve based on all incoming neighbors, then eventually each will have as its estimate the actual minimum-weight of a path from v_0 to itself.

The second is a termination property, and invariants aren't much help in proving termination.
There are other methods for this, "beyond the scope" of this course.
So we only concentrate on the first property.
This is proved as an invariant:

Invt1: For any v, if $pweight(v) \neq \infty$, then $pweight(v)$ is the total weight of some path from v_0 to v.

Prove by induction as usual.
BASE:
Only non-∞ node is $v0$, and there is a path of total weight 0, namely trivial 0-length path.
INDUCTIVE STEP:
By inductive hypothesis, $pweight(w)$ before the step is the total weight of some path from v_0 to w.
Augment this path with one additional edge to get a path to v.

5 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.

Give two examples, one simple, one not so simple.
Will represent each as a single state machine.

5.1 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.

Do more carefully in recitation?

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).

Do in recitation?

5.2 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 is 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

Transitions:

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

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

$receive(m,b)_{1,2}$

Pre:
$(m, b) = head(channel_{1,2})$
Eff:
remove *head*
if $b \neq tag_2$ then
add m to buf_2
$tag_2 := tag_2 + 1 \bmod 2$

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

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); talk through.

Also need:

Invt3:
If (m, tag_1) is in $channel_{1,2}$ then $m = head(\mathit{work\text{-}buf}_1)$.

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:
$buf_2 \cdot \mathit{work\text{-}buf}_1 = buf_1$.
2. If $tag_1 = tag_2$ then
$\mathit{work\text{-}buf}_1$ and buf_2 are nonempty,
$head(\mathit{work\text{-}buf}_1) = last(buf_2)$, and
$buf_2 \cdot tail(\mathit{work\text{-}buf}_1) = buf_1$.

tail = all but head.

Invt4, part 2, easily implies the goal Invt1.
Also implies that $\mathit{work\text{-}buf}_2$ is always nonempty when the $receive_1$ occurs for equal tags.
(So parenthetical check is always true.)
Prove Invt4 by induction.
BASE:
tags unequal, $\mathit{work\text{-}buf}_1 = buf_1$ 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 $\mathit{work\text{-}buf}_1$, by Invt3.
Change tag to make equal, replicate $first(\mathit{work\text{-}buf}_1)$ 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.

Study problems on state machines:

1. Consider a state machine whose state consists of:
V, a set of vertices, initially containing one vertex.
E, a subset of $V \times V$, initially empty
$n \in \mathbb{N}$, initially 1.

Transitions:
$expand(v, w, x)$ Precondition:
$v \in V$
there is at most one edge incident on v in E
$w, x \notin V$
Effect:
$V := V \cup \{w, x\}$
$E := E \cup \{(v, w), (v, x)\}$
$n := n + 2$
Prove the following invariants (in order):
1. $|V| = n$
2. (V, E) is a tree 3. if $|V| \geq 2$ then (V, E) has $\frac{n-1}{2}$ internal nodes.

2. (Binary search) Fix k.
Consider a state machine whose state consists of:
a, an array indexed by $\{1, \ldots, 2^k - 1\}$ of $\mathbb{N}$, initially arbitrary, subject to $a(i) \leq a(j)$ if $i < j$ (i.e., sorted).
$v \in \mathbb{N}$
$i \in \mathbb{N}$, initially 2^{k-1}
$jump \in \mathbb{N}$, initially 2^{k-1}
$ans \in \{1, \ldots, 2^k - 1\} \cup \{unknown, no\}$, initially $unknown$

Transitions:

$less$
Precondition:
$ans = unknown$
$v < a(i)$
Effect:

if $jump > 0$ then
$jump := jump - 1$
$j := j - 2^{jump}$
else $ans := no$

$greater$ Precondition:
$ans = unknown$
$v > a(i)$
Effect:
if $jump > 0$ then
$jump := jump - 1$
$j := j + 2^{jump}$
else $ans := no$

$equals$ Precondition:
$ans = unknown$
$a(i) = v$
Effect:
$ans := i$

Prove invariants: 1. If $ans = no$ then $(\forall i) v \neq a(i)$ 2. If $ans = i \in \mathbb{N}$ then $a(i) = v$ (Hint: You will need to strengthen these invariants in order to prove them.)