Notes on Computational Processes, Installment 2

Nancy Lynch*

February 28, 2000

1 Overview

Last time introduced the general idea of a computational process, modelled mathematically as a state machine.

Did examples, based on little (nondeterministic) games, and (mainly deterministic) sequential algorithms.

Today do one more type of example—a state machine used as an abstract description of a complex system.
(Nondeterministic).

2 Abstract system descriptions

2.1 Overview

A caching system, may be used in a multiprocessor machine or a distributed system.

Informally:
Have a "memory", which contains the latest versions of all data in some application.
Also have one or more "caches", which contain some versions of some of the data.
These caches are intended for fast access by individual clients.

Examples: Web pages. Numerical data values in a multiprocessor.

Model uses two basic sets: *addresses*, *values*

The system contains:
memory, a (total) function from *addresses* to *values*
cache1, a function from *addresses* to *values* $\cup$ $\{null\}$
cache2, ...

The two caches are supposed to be used by two different clients, client 1 and client 2.

2.2 Weak caching system

What we want to express in the model:
The memory can be updated at any time.
This may leave one or both of the caches out-of-date.
Also, at any time, the latest value may be copied from the memory to a cache, overwriting any older value that's there.
Also, at any time, a value may be dropped from a cache.

As a state machine:

Q: $\{(memory, cache1, cache2)$ of the types described above$\}$.
Q_0: memory is arbitrary, caches map everything to *null*.
δ:
$(memory, cache1, cache2 \rightarrow (memory', cache1', cache2')$
where (five kinds of steps):

1. New memory is same as old memory except that at most one of the addresses gets a new value. Caches unchanged.

$\exists a \in addresses, \forall b \in addresses - \{a\}[memory'(b) = memory(b)]$,
$cache1' = cache1$, and $cache2' = cache2$

Actually, it's probably easier to write this in assignment form, which you just say what's changed:

$memory(a) :=$ arbitrary *value* (not null).

2. One address in *cache1* gets updated from memory.

$cache1(a) := memory(a)$

3. One address in *cache2* gets updated from memory.

$cache2(a) := memory(a)$

4. One value gets dropped from *cache1*.
$cache1(a) := null$

5. One value gets dropped from *cache2*.

$cache2(a) := null$

Nondeterministic.

What would one prove about this service?

Basic correctness:
Maybe something about the memory having the most up-to-date value.
LTTR

Eventuality:
Must be very careful here.
Tricky, and we won't go into this carefully in this course.

E.g., we might want to say that the caches eventually get values that are put (and stay) in memory.
But to say this, we must first say which steps must happen.

For sequential programs, I used the notion of "live" execution to express this; it says that something keeps happening if anything is enabled.
But that general definition generally doesn't give what we want for service models.
E.g., we'd like to allow the case where no new memory updates happen.
And we'd like to disallow the case where memory updates keep happening and the caches never get updated.

Saying these kinds of things is a little complicated, and I won't go into it here.

2.3 Strong caching system

The spec given so far allows clients to obtain out-of-date information.
E.g., memory value updated, but client still has old value.
Sometimes we want to make sure the clients are never out-of-date.

One thing we could do is make sure that, when the memory is updated, the caches are cleaned out.

This just involves modifying the memory update step a little:
$memory(a) :=$ arbitrary $value$ (not null).
$cache1(a) := null$
$cache2(a) := null$

Now get a stronger guarantee
Intuitively, clients never see old values.
Can show, in all reachable states:

Either $cache1(a) = memory(a)$ or $cache1(a) = null$

In other words:
If $cache1(a) \neq null$ then $cache1(a) = memory(a)$.

And likewise for *cache2*.

Also, could mix and match—some weak and some strong caches.
The idea here is that the clients could say if they care whether their values are always up-to-date.
Of course, if they say yes to this, they may sometimes have to wait to get the new value.

3 Kinds of properties proved for computational processes

We've now seen many examples of computational processes, modeled as state machine.
Now let's take a closer look at the kinds of properties we proved about the computational processes.
Some of these fall into certain standard categories:

What?

1. Some predicate (of the state variables) is true in all reachable states.

Examples:

Die Hard: The contents of each jar is a multiple of 3 gallons.

2-dimensional puzzle: $x + 2y$ is a multiple of 7.

GCD: $GCD(x, y) = GCD(a, b)$.

Graph coloring:
No 2 neighbors are ever colored the same color.

Strong caching:
If $cache1(a) \neq null$ then $cache1(a) = memory(a)$.
If $cache2(a) \neq null$ then $cache2(a) = memory(a)$.

2. Eventuality properties

Examples:

GCD: Eventually get $x = y$.

Graph coloring:
Eventually color all the vertices.

Relation transitive closure:

Eventually get T equal to the transitive closure of R.

But note that some examples don't have termination properties, e.g., Die Hard may go on forever.
Expect cache example to go on forever.

3. Time bound properties

Examples:
Relation transitive closure: After $\lceil \log n \rceil$ steps, T is the transitive closure of R.

These kinds of properties are so common that people have developed stylized ways of stating and proving them, languages for expressing them, computer tools for checking them,...

The most important one is the first.
Called *invariants*.

4 Invariants and the Invariant Theorem

4.1 Invariants

Definition:
An *invariant* of a state machine $M = (Q, Q_0, \delta)$ is a predicate $P(q)$ that is true for all reachable states q.

Writing this as a predicate of the state.
In practice, it is written as a predicate of the variables in the state.

Example: 2-dimensional puzzle:
The invariant "$x + 2y$ is a multiple of 7" is a predicate of the state (statement about a particular state); it is expressed in terms of the values of the variables x and y in the state.

As we have seen, many important properties of sequential algorithms, games, and other computational processes can be expressed as invariants.
Therefore, proving invariants is an important part of verifying the correctness of programs, systems,...

4.2 Theorem

Let's describe a specific method for proving an invariant.

How did we prove the ones we talked about yesterday?

The basic strategy was induction on the number of steps in a finite execution, saying that the predicate is true after that number of steps.
Actually, we usually had to first strengthen the predicate to something for which the induction would work.

Started with $n = 0$.
Used ordinary induction, not strong induction.

We can say more about the proofs:
The base case, corresponding to 0-step executions, always amounted to proving P for the initial states.
The inductive step, going from n steps to $n+1$ steps, always amounted to checking that every step "preserved" the property, that is, that if the property was true before the step, it was also true after the step.
This in turn broke down into a proof by cases.

The following proof outline expresses this general strategy (without the strengthening part):

[[[Victor: slide]]]

Given: $M = (Q, Q_0, \delta)$ is a state machine.
Prove: P is an invariant of M.
1. $\forall q \in Q_0(P(q))$
2. $\forall (q, q') \in \delta(P(q) \Rightarrow P(q'))$.
3. QED Invariant theorem

Now, what's the "invariant theorem"?
It's just a theorem that says that showing these two conditions is enough to imply that P is an invariant.

Theorem 4.1 (Invariant theorem)
Suppose $M = (Q, Q_0, \delta)$ is a state machine, and P is a predicate of states in Q such that:

1. $\forall q \in Q_0(P(q))$, and
2. $\forall (q, q') \in \delta(P(q) \Rightarrow P(q'))$.

Then P is an invariant of M (that is, is true in all reachable states of M).

Prove this using ordinary induction, on the number of steps in the execution:
First formulate a new predicate $P'(n)$ that asserts P for all the states that can be reached by n-step executions.
Prove P' using ordinary induction on n.

The base step, $P'(0)$, says that P is true of all states that occur in 0-step executions, that is, the start states.
That follows easily by assumption.
The inductive step takes a little more work, but basically follows from the second assumption.

Given: $M = (Q, Q_0, \delta)$ is a state machine.
Given: $\forall q \in Q_0(P(q))$
Given: $\forall (q, q') \in \delta(P(q) \Rightarrow P(q'))$.
Prove: P is an invariant of M, that is, $\forall q \in Q(q \text{ reachable } \Rightarrow P(q))$
1. $\forall n \geq 0(P'(n))$, where $P'(n)$ is the predicate $\forall q$, a final state of an n-step execution $(P(q))$.
    1. (Base) $P'(0)$ — By the assumption about start states
    2. (Inductive step) $\forall n \geq 0(P'(n) \Rightarrow P'(n+1))$
        1. Fix $n \geq 0$
        2. Assume $P'(n)$
        3. $P'(n+1)$ — ???
        4. QED
    3. QED — Induction
2. QED — Definition of "reachable"

Expanding the inductive step shows how the second assumption is used.
Namely, we consider the final state of any $n+1$-step execution, in order to show that P holds there.
We see that this arises from the final state of an n-step execution, via one more step.
But the state after the n-step execution satisfies P, by inductive hypothesis.
And the last step preserves P, by assumption.

3. $P'(n+1)$, that is, P is true for all final states of $n+1$-step executions.
    1. Fix an $n+1$-step execution $q_0, q_1, \ldots, q_n, q_{n+1}$.
    2. q_n is the final state of an n-step execution.
    3. $P(q_n)$ — Inductive hypothesis (1.2.2)
    4. (q_n, q_{n+1}) is a step of M.
    5. $P(q_{n+1})$ — By the assumption about steps preserving P
    6. QED — UG

This proof may look a little confusing because there are *two* predicates involved, P and P'. Just remember that P is a predicate about states, while P' is a predicate about numbers (lengths of executions).

4.3 Examples from last time

The theorem is pretty abstract, but it's easy to apply.
First revisit a couple of the examples from the previous lecture:
Die Hard: $P(q)$: The contents of each jar is a multiple of 3 gallons.

The proof just does the minimum work:
shows P holds in the start state and is preserved by all steps (cases):

Prove: P is an invariant of the Die Hard state machine.
1. $\forall q \in Q_0(P(q))$ — P true for start state $(0, 0)$.
2. $\forall (q, q') \in \delta(P(q) \Rightarrow P(q'))$.
    1. Fix $(q, q') \in \delta$
    2. Assume $P(q)$.

3. $P(q')$
    1. If (q, q') fills the little jar, then $P(q')$.
    2. If (q, q') fills the big jar, then $P(q')$.
    3. If (q, q') empties the little jar, then $P(q')$.
    4. If (q, q') empties the big jar, then $P(q')$.
    5. If (q, q') pours from little to big, then $P(q')$.
    6. If (q, q') pours from big to little, then $P(q')$.
    7. QED   Cases
4. QED   Implication, UG

3. QED   Invariant theorem

Really the same argument as before, but the previous proof said more, e.g., about the underlying induction.

Strong caching:
$P(q)$: In state q: If $cache1(a) \neq null$ then $cache1(a) = memory(a)$.
If $cache2(a) \neq null$ then $cache2(a) = memory(a)$.

Prove: P is an invariant of the strong caching state machine.

1. $\forall q \in Q_0(P(q))$ P true for start state, where both caches are empty.
2. $\forall (q, q') \in \delta(P(q) \Rightarrow P(q'))$.
    1. Fix $(q, q') \in \delta$
    2. Assume $P(q)$.
    3. $P(q')$
        1. If (q, q') puts a new value in memory, then $P(q')$.
           Makes the caches null for the affected address.
        2. If (q, q') updates a cache from memory, then $P(q')$.
           Makes cache = memory, at the affected address.
        3. If (q, q') drops a value from a cache, then $P(q')$.
           Makes cache null.
        4. QED   Cases
    4. QED
3. QED   Invariant theorem

5 Stable marriage problem

Another algorithm (or is it a game) example:
Not obvious at all.
Can use invariants to help understand.

n boys and n girls in the Ballroom Dance club at MIT.
Want to pair up, boy-girl only.

So, this being MIT, decide to use math to help set up the pairings.
Each boy ranks all girls, and vice versa, according to preferences.

No ties allowed.

What's a good pairing?
One that's *stable*, in that it avoids *rogue pairs* (unstable pairs):
a situation where two people, b and g, prefer each other to their own partners.

That's unstable, for obvious reasons—b and g will desert their own partners for each other.

Not obvious how to get a stable pairing.
In fact, not even obvious one always exists.

Example:

Boys' preferences:
John: Ally, Bea, Cass, Dee
Kyle: Dee, Bea, Cass, Ally
Len: Ally, Dee, Bea, Cass
Matt: Bea, Ally, Dee, Cass

Girls' preferences:
Ally: Kyle, Len, Matt, John
Bea: Kyle, Len, Matt, John
Cass: Kyle, Matt, John, Len
Dee: Len, Kyle, John, Matt

Try greedy algorithm:
Go down list of boys letting each pick favorite among still available girls:

John picks Ally,
Kyle picks Dee,
Len picks Bea,
Matt picks Cass.

Rogue pair: *Ally* and *Len*:
Ally prefers *Len* to her actual partner John.
Len prefers *Ally* to his actual partner *Bea*.

So, I'll present a strategy that does work.

Asymmetric:
The boys pursue the girls, and the girls send away their least favorites from among those that are pursuing them.

Works the other way too.

A little more precisely:
The girls hang around in various places around the gym.

Each boy carries around his list of girls, goes to talk to the first girl on his list.
Any girl who has more than one boy hanging around her can tell any except her favorite one of these to go away.
That makes him give up, cross her off his list, and go on to talk to the next girl on his list.
And so on.
(If he runs out of girls, he just goes home and plays Doom.)
Continues until no girl has more than one boy hanging around.

Example:
Try the example earlier. Initially:
John and Len pursue Ally,
Kyle pursues Dee,
Matt pursues Bea.

Ally has two boys hanging around, prefers Len, tells John to go away. He goes on to Bea.
Now:
Len pursues Ally,
Kyle pursues Dee,
Matt, John pursue Bea.

Bea tells John to get lost.
Now John goes on to Cass.
Now:
Len pursues Ally,
Kyle pursues Dee,
Matt pursues Bea,
John pursues Cass.

and no girl tells anyone to go away.
Claim no rogue pairs.
Why?
Well, three boys have their first choices, so they can't be part of rogue pairs.
John has his third choice, so conceivably he could be part of a rogue pair.
That would mean that he would prefer someone other than the person he has (Cass), and that someone would also prefer him to her partner.

But the only girls he prefers to Cass are the two who already rejected him.
And the fact that they rejected him means that, at that time, they had better choices.
Moreover, their choices never got worse, because they always keep their best choices.
So they still don't prefer John.

We can generalize these observations.
Namely, we can show:

The algorithm always terminates.
The resulting pairing pairs up everyone.

Pairing is stable.

In fact, can show more—boys get the best possible match, girls get the worst!
Strategy favors the pursuers.
But leave this for pset.

In this section, we'll say what we can about this using invariants.
Turns out we can say quite a lot.

Model as state machine:
What information is needed to describe the state of the algorithm?
It's enough to know what number each boy is up to on his list.
So, for each boy b, the state contains $\textit{next-number}(b)$, which gives the number of the girl on his list that he is currently pursuing. If he should happen to use up the whole list, his *next-number* will get set to $n+1$ as a default value.

Q:
For every boy b, a value $\textit{next-number}(b) \in \{1, 2, \ldots, n+1\}$.

In the initial state, all boys pursue their first choices:

Q_0:
$\textit{next-number}(b) = 1$ for all b.

There is only one kind of transition:
A girl rejects a boy, and he goes on to the next girl on his list:

δ:

A step of the form "g rejects b" can occur only if:
b is currently pursuing g,
another b' is currently pursuing g,
and g prefers b' to b.

When it occurs, the effect on the state is (just) that $\textit{next-number}(b) := \textit{next-number}(b) + 1$.

How to express the "enabling condition" more precisely?
"b is currently pursuing g" means that:
$\exists i(\textit{next-number}(b) = i \wedge g = \textit{girl}(b, i))$.

Here, I'm writing $\textit{girl}(b, i)$ for the ith girl on b's list.

Similarly, express b' is pursuing g.

g's preference says:
$\exists i, i'(b = \textit{boy}(g, i) \wedge b' = \textit{boy}(g, i') \wedge i' < i)$.

What kinds of properties might we want to know about this algorithm, that might be expressed in the form of invariants?

1. No boy ever runs out of choices.

Expressed as an invariant:
$\forall b(\textit{next-number}(b) \leq n)$

2. If no further moves are possible, then the boys and girls are all paired up.

Dissect this a bit.
"No further moves possible" just means that no girl has two or more pursuers.
That is, every girl has 0 or 1 pursuers.

"Boys and girls are all paired up" means that every girl has exactly one pursuer.

So this amounts to saying: If every girl has 0 or 1 pursuers then every girl has exactly one pursuer.
Could express in terms of *next-number* and the rank lists (but I won't write this out).

3. If the boys and girls are all paired up, then there are no rogue pairs.

If every girl has exactly one pursuer, then there do not exist b, g, b', g', where b' pursues g, b pursues g', g prefers b to b' and b prefers g to g'.

Can't just prove these invariants directly using the Invariant Theorem (which amounts to proving them by induction).
Need to say something a little deeper about why the algorithm works right.
If we say the right thing, it should (we hope) be provable using the Invariant Theorem.

Look at the last statement—why can't this arise?
Suppose it did.
Then b is pursuing g', although he prefers g.
That means he must have previously been rejected by g.
Why did g do this?
Because she found someone she liked better.
But then she must *still* have someone she likes better (because she never sends her best choice away).
So in the end, when she has only one pursuer b', he must be someone she prefers to b.

This argument is a little intricate, and a little hard to get right.
Can formulate the key idea using an invariant that can be proved with the invariant theorem.
(I'm making a bit of a jump here, pulling this out of thin air. But it comes from considering arguments like the one I just made.)

Invariant 0:
(In any reachable state)
If b has crossed g off his list,
then g is pursued by someone that g prefers to b.

Make sure you know how to expand this into formal statement.
E.g., the hypothesis means:
"*next-number*(b) is greater than the rank b assigns to g".
Could be pursuing another girl, or might have run off the list.

Turns out we can prove Invariant 0 using the Invariant Theorem:

Define $P(q)$:
In state q, for all b, g, if b has crossed g off his list, then g is pursued by someone that g prefers to b.

The argument about the start state is easy (as usual).

Prove: P is an invariant of the matching state machine.

1. $\forall q \in Q_0(P(q))$	Vacuously true—everyone is pursuing his top choice.
2. $\forall (q, q') \in \delta(P(q) \Rightarrow P(q'))$.	
1. Fix $(q, q') \in \delta$	
2. Assume $P(q)$.	
3. $P(q')$	???
4. QED	
3. QED	Invariant theorem

The argument about steps again breaks down into cases.
But what cases?
This gets a little tricky.

The only kind of step is one where some girl rejects some boy.
But the statement we're trying to prove is about *all possible* boys and girls.
So, the interesting cases involve things like whether the girl and boy about whom we're trying to prove the property are the same ones or different ones from the ones involved in the rejection step.

Fix b, g (the ones we want to prove the property for).
For emphasis, call them Bob and Gail.

The step we're considering involves some girl rejecting some boy.
Cases:
1. Gail rejects Bob.

This is potentially dangerous, because now Bob, for the first time, crosses Gail off his list.
Makes the hypothesis of the statement we're trying to prove true.

We must show the conclusion is true after the step.
Clearly Gail is pursued by someone she likes better, because that's why she chased Bob away.

2. Another girl rejects Bob.

We use the fact that P is true before the step to see that it's still true after the step.
If Bob has already crossed Gail off after the step, then he already had done this before the step.
Since P was true before the step, Gail must have had a pursuer she preferred to Bob, before the step.
She still has him after the step.

3. Gail rejects another boy.

Can't make P false if it was previously true.
Doesn't affect whether Bob has crossed Gail off (can't make the hypothesis true).
Also, Gail doesn't send away her best choice, so can't make the conclusion false.

4. Another girl rejects another boy.

Can't make P false if it was previously true.
It doesn't make Bob cross off Gail.
And it doesn't remove any of Gail's pursuers.

Written in a more stylized way:

3. $P(q')$, that is, $\forall b, g((b$ has crossed off g in $q') \Rightarrow (g$ has a pursuer she prefers over b in $q')$
    1. Fix Bob, Gail.
    2. Assume Bob has crossed off Gail, in q'.
    3. Gail has a pursuer she prefers over Bob, in q'.
        1. If in (q, q'), Gail rejects Bob, then Gail has a pursuer she prefers over Bob, in q'.
           Definition of the step.
        2. If in (q, q'), another girl rejects Bob, then Gail has a pursuer she prefers over Bob, in q'.
           ???
        3. If in (q, q'), Gail rejects another boy, then Gail has a pursuer she prefers over Bob, in q'.
           ???
        4. If in (q, q'), another girl rejects another boy, then Gail has a pursuer she prefers over Bob, in q'.
           Doesn't affect the truth of P.
        5. QED — Cases
    4. QED — Implication, UG

And the two ??? cases:

2. If another girl rejects Bob, then Gail has a pursuer she prefers over Bob, in q'.
    1. Assume another girl rejects Bob in (q, q').
    2. Bob has crossed off Gail in q. — 2.3.2, doesn't cross off Gail in this step.

3. Gail has a pursuer she prefers over Bob, in q.
   By assumption $P(q)$ (2.2)
4. Gail has a pursuer she prefers over Bob, in q'.
   Gail's pursuers don't change.
5. QED — Implication

3. If Gail rejects another boy, then Gail has a pursuer she prefers over Bob, in q'.
   1. Assume Gail rejects another boy in (q, q').
   2. Bob has crossed off Gail in q. — 2.3.2, doesn't move in this step.
   3. Gail has a pursuer she prefers over Bob, in q.
      $P(q)$ (2.2).
   4. Gail has a pursuer she prefers over Bob, in q'.
      Gail doesn't send away her top-ranked pursuer.
   5. QED — Implication

It turns out that, with Invariant 0, the other invariants we mentioned earlier are easy to prove.

Invariant 1:
No boy ever runs out of choices.
That is, $\forall b(\textit{next-number}(b) \leq n)$

Proof:
By contradiction.
Assume that, in some reachable state q, and for some b, $\textit{next-number}(b) = n + 1$.
Then (in q), b has crossed off all the girls.
So by Invariant 0, every girl has a current pursuer that she prefers to b.
That's a total of at least n pursuers, plus b, or at least $n + 1$ boys.
But there are only n total boys.
Contradiction.

Invariant 2:
If every girl has at most one pursuer, then every girl has exactly one pursuer.

Remember, this says if no further moves are possible, then everyone is paired up.

Proof: Assume every girl has at most one pursuer.
Every boy is pursuing some girl, by Invariant 1.
So, n pursuers.
No two boys pursue the same girl.
So all n girls must have pursuers (one each).

We already know that, if the algorithm ever reaches a point where no more steps are

enabled, then everyone is paired up.
It remains to show that such a final pairing is stable (has no unstable pairs).

Invariant 3:
If the boys and girls are all paired up, then there are no unstable pairs.

Proof:
By contradiction.
Suppose in state q, everyone is paired up, and there is an unstable pair (b, g).
That is, b is paired with some g', g with some b', b prefers g to g', g prefers b to b'.

Since b prefers g to his own partner (the one he's pursuing, in state q), Invariant 0 implies that, in q, g has some pursuer that she prefers over b.
But g's only pursuer is b'.
Therefore, she prefers b' to b, a contradiction.

It is possible to take this example further, to show something about the fairness of this algorithm to boys vs. girls.
Who do you think does better?

Turns out that the boys do.
In fact, it is possible to show that the boys do as well as they possibly could.

Meaning:
If a boy b prefers some girl to the girl he ends up with in this algorithm, then she isn't even possible for him:
she can't be his partner in *any* stable pairing.

On the other hand, the girls do as badly as they possibly could.
If a girl likes some boy less than the one she ends up with, then he isn't even possible for her.

The moral is that being the pursuer pays off.

Internship matching algorithm.
Favors the interns.

Have still not proved that the algorithm actually terminates.
That is, every live execution is finite.
(If it keeps taking steps as long as any are enabled, then eventually, it will reach a point when no more steps are enabled.) Why is this true?

At every step, some boy increases his *next-number* (and the others stay the same).

This can't go on forever.
In fact, each boy gets rejected at most $n-1$ times, for a total of at most $n(n-1)$ rejection steps.

Treat termination a little more systematically in the next section.

6 Termination Theorem

We can formulate a simple theorem describing a proof method for showing termination.
The method basically involves associating a natural-number measure with each state, in such a way that the measure decreases with each step.
Because the natural numbers stop at 0, these decreases can't go on forever.

This method is just used to show that there are no infinite executions, that is, in any sequence of steps, eventually a point will be reached where no further steps are possible.

That doesn't automatically say that the problem solution has been produced, though.
Another step is needed in applying this method—something to say that when no further steps are possible, the problem has actually been solved.

6.1 The theorem

Definition:
Suppose $M = (Q, Q_0, \delta)$ is a state machine.
Then a *termination function* for M is a function f mapping reachable states of M to N, such that for every reachable q and q' with (q, q') in δ, $f(q') < f(q)$.

Emphasize strictly less.

Theorem 6.1 (Time bound theorem)
Suppose $M = (Q, Q_0, \delta)$ is a state machine, and $f : Q \rightarrow N$ is a termination function for M.
Then the number of steps in any execution starting from $q \in Q_0$ is at most $f(q)$.

Proof. By contradiction.
Suppose this bound does not hold, and choose an execution $q = q_0, q_1, \ldots, q_t$, $t > f(q)$.
Then $f(q_t) \leq f(q_0) - t$ (this can be shown by induction on the length of a prefix of the sequence).
But the right-hand side is strictly less than 0, which implies that $f(q_t) < 0$.
Contradiction, since $f(q_t)$ is a natural number. ∎

The previous theorem gives a time bound for executions starting in each possible start state.
The following corollary talks about all executions of the machine.
But it doesn't give a specific time bound—it just says it eventually terminates.

Theorem 6.2 (Termination theorem)
Suppose $M = (Q, Q_0, \delta)$ is a state machine, and $f : Q \to N$ is a termination function for M.
Then every execution of M is finite.

Proof. By contradiction.
Suppose there is an infinite execution.
Fix one; it begins in some start state q.
By the time bound theorem, the length of the execution must be finite (as most $f(q)$).
Contradiction. ∎

Written as structured proofs:

Given: $M = (Q, Q_0, \delta)$ is a state machine.
Given: e is an execution of M, $k \in N$.
Prove: e has at most k steps.
1. $f : Q \to N$ is a termination function for M.
2. $f(q) \leq k$, where $q \in Q_0$ is the first state of e.
3. QED — Time bound theorem

Given: $M = (Q, Q_0, \delta)$ is a state machine.
Prove: Every execution of M has at most k steps.
1. $f : Q \to N$ is a termination function for M.
2. $\forall q \in Q_0 (f(q) \leq k)$
3. QED — Time bound theorem

Given: $M = (Q, Q_0, \delta)$ is a state machine.
Prove: Every execution of M is finite.
1. $f : Q \to N$ is a termination function for M.
2. QED — Termination theorem

6.2 Examples

6.2.1 Stable marriage problem

Define a termination function.
Recall we said that some *next-number*(b) increases at each step.
Suggests we look at Σ_b*next-number*(b).
Almost right—but that's increasing, not decreasing.

So turn it around, defining $f(q)$ by:
$\Sigma_b(n - \textit{next-number}(b))$.

How do I know this function is defined?
Could we have *next-number*$(b) > n$?
That could make the function go negative...

But notice that the function only has to be defined for the *reachable* states.
And Invariant 1 says that $\mathit{next\text{-}number}(b) \leq n$ in all the reachable states.

I could have been more conservative, setting the inner term to be $(n + 1 - \mathit{next\text{-}number}(b))$.
But that would give me a worse time bound result.

Given: M is the matching state machine.
Given: $f(q)$ is defined as $\Sigma_b(n - \mathit{next\text{-}number}(b))$.
Prove: Every execution of M has at most $n(n-1)$ steps.
1. f is a termination function for M.
    1. f is defined on all reachable states. Invariant 1
    2. $\forall q, q'$, reachable states with $(q, q') \in \delta(f(q') < f(q))$ Definition of step.
2. $\forall q \in Q_0(f(q) \leq n(n-1))$ It's equal, because all boys start with 1
3. QED Time bound theorem

What has this given us?
We know that every execution ends after at most $n(n-1)$ steps (in particular, it eventually ends).

But that doesn't by itself say that it produces a matching.
To see this, we have to go back to our invariants.

The missing piece is Invariant 2, which said:
"If no further moves are possible, then everyone is paired up."

Coupling this with the fact that the execution eventually reaches a point where no further moves are possible, this sets that eventually (actually, within $n(n-1)$ steps), everyone gets paired up.

6.2.2 GCD

Define termination function:
$f(q)$: $x + y$
Defined on all (reachable) states, decreases with every step.
So, termination theorem says that the algorithm eventually terminates.

For any particular execution, can get a time bound of $a + b$, the sum of the original inputs.

But remember, this just says the execution has to eventually stop.
The missing piece: If no more steps are enabled, the problem has been solved.
Here, no more steps enabled means $x = y$.
We've already argued using invariants that if $x = y$ then x is the correct GCD.
Coupling this with eventually $x = y$ says that eventually x is the correct GCD.

6.2.3 Graph coloring

Define termination function:
$f(q)$: Number of uncolored vertices.
Defined on all reachable states, decreases with each steps, so eventually no more steps are enabled.

Separately argue that if no steps are enabled, every node has been colored (argued using invariants).
Together this says that eventually every node gets colored.