Notes on Computational Processes, Installment 3

Nancy Lynch*

February 27, 2000

1 Overview

We've seen the basic idea of a computational process and how to model it.
Computational processes include simple (nondeterministic) games, ordinary sequential programs, abstract descriptions of system services.

Also include more complex systems.

Examples:
Computer operating systems
World-Wide Web.

However, such systems are just too big to model as one big, complicated state machine.
We need ways of breaking the descriptions into pieces.
Structure.

What kind?
Two main methods are used:
Abstraction, composition.

Abstraction:

This refers to giving high-level descriptions of systems, separate from the detailed implementations.
Can use high-level data types, like relations, graphs, trees,
rather than (or in addition to) low-level types, like numbers, Booleans, and strings.
Use high-level operations, e.g., taking transitive closure of a complicated relation, coloring a graph.
Can reason about the system's behavior in terms of such high level data types and operations.

Then separately, implement the high-level data types and operations in terms of available low-level data types and operations.
Now just have to convince ourselves that the implementation correctly implements the high-level description,
but don't have to think anymore about correctness of the high-level description.

Have broken down both the implementation and the reasoning about its correctness.

Composition:

This refers to building a system up out of individual components.
Build the components separately, give specifications for what each one does, reason separately about how each component implements its specification.

These kinds of system decomposition are very common in practical programming.
What is not so common is very careful reasoning about the correctness of the decomposed implementations.
Usually, reasoning about correctness of complex systems is done fairly informally.
But ideally, we could be quite rigorous, e.g., doing really proofs about system behavior.

To do this, we need the right math to support it.

What math?
The basis is just what we've been studying, state machines to model computational processes.

For system decomposition, need a little more:
Need to capture the idea that some parts of a computational process (component) are intended to be *externally visible*, so they can be used by people or by other programs.
On the other hand, other parts of a computational process are only used for the implementation, and don't need to be seen from the outside.
In fact, it's better that they aren't seen from the outside, because that would make the process look at lot more complicated.

Separating external and internal information in a computational process is useful for modeling abstraction and composition.

E.g., if we want to say that a complex implementation "correctly implements" a simple abstract component description, we have to say what we mean by "implements".
In particular, what is "the same" about the implementation and the abstract specification?
The externally-visible behavior.

E.g., when you build a system out of lots of components, you want to think about the external behavior of the components, not about the details of how all the components are implemented.

In the rest of this lecture, I will:
1. Define externally-visible behavior of a state machine (and give examples).
2. Define abstraction for state machines (and give examples). Maybe a theorem.
3. Say just a little, informally, about composition.

2 Externally-visible behavior

2.1 GCD example

GCD algorithm, using subtraction.
Started with a, b in the state, ended up with x in the state; we knew it was the answer when $x = y$.

Modify this so the a and b come in to the algorithm as inputs, and the GCD is provided by the algorithm as an output.
(This time I'm not bothering to remember the a and b in the state forever—that was useful for proving an invariant, but it's not actually needed for the algorithm.)
Formally:

$Q = \{(x, y) \in N^+ \cup \{null\}$.
Q_0: Initially both null.

δ:
Has three kinds of steps: input steps, output steps, and internal (computation) steps:
$(x, y) \to (x', y')$, where:

Input steps:
request-GCD(a, b) steps:
If $x = y = null$ then $x' = a$ and $y' = b$, else $x' = x$ and $y' = y$.

Computation steps:
compute:
Either $x > y$, $y' = y$, and $x' = x - y$,
or $y > x$, $x' = x$, and $y' = y - x$

Output steps:
return-GCD(r):
$x = y = r$, $x' = y' = null$

Thus, an initial input just sets x and y to start things off.
As a technicality, this says that if other inputs arrive while a GCD computation is going on, they just get ignored.
Computation is as before.
When the computation is done, an output can be performed, after which the x and y variables go back to null values.
(Thereafter, more GCD computations could be handled by the same process.)

The labels *request-GCD*$(a, b), a, b \in N^+$, *compute*, *return-GCD*$(r), r \in N^+$ are actually part of the model.

We will augment the definition of a state machine to allow "labels" for steps (below).

Also, augment the definition of an execution to include the labels.

Example:
An execution of the labeled state machine for GCD:
$(null null)$,
request-GCD$(9, 6)$,$(9, 6)$,
compute,$(6, 3)$,
compute,$(3, 3)$,
return-GCD(3), $(null, null)$.

External behavior:

Now, we want to model the externally-visible behavior of this process.
Can reasonably say that what the user sees is just the input he submits and the return values he gets back.
Model this by extracting the input and output labels from an execution.
Suppress internal labels and states.
What's left is called a "trace".

Example:
From the execution above, we can extract the trace:
request-GCD$(9, 6)$, *return-GCD*(3).

Notice that this trace only tells us the input and output—it says nothing about how the process actually computed the GCD (by repeated subtraction steps).

What might we prove about executions of this labeled state machine?
E.g., look at the simple case of exactly one input:

If an execution contains exactly one *request-GCD* label, then:
– It contains at most one *return-GCD* label.
– If it contains a *return-GCD*(r) label, then r is the GCD of the arguments in the *request-GCD* label.

Could also prove an eventuality property, saying that the answer is eventually produced.
But this is a little tricky:
We have to hypothesize a live execution in order to talk about eventuality.
But liveness is a little tricky to define in the presence of inputs and outputs.
I won't go into this here, in fact, won't mention liveness again today.

Notice that these properties only talk about externally-visible behavior.
Many interesting properties of processes are formulated that way.

Example:

Recall that, a couple of weeks ago, we presented an alternative version of the GCD algorithm, where at each step, instead of subtracting $x - y$ (or $y - x$) to get a new value for the next step, we let the new value be the remainder on division of x by y (or y by x).

Could formulate this as a labeled state machine, very similar to the one for the subtraction-based GCD.
All that changes is the *compute* steps.
Although executions of this new labeled state machine are different from those for the other one, the two machines have the same traces.
Thus, in a sense, they are "equivalent", as far as a user could tell.

(Could have different performance—we're just talking about correctness here.)

2.2 Labelled state machines—Definitions

Now we can generalize what we did for the GCD state machine, to give a general definition of labeled state machine, and its executions and traces.

A *labeled state machine* consists of:

(Q, Q_0, δ, L), where:

Q, Q_0 are states and initial states, as before.
L is a set of *labels*, partitioned into input, output and internal labels.
$\delta \subseteq Q \times L \times Q$.

Thus, every step is labeled with a "label" (element of L), which may be input, output or internal label.

An *execution* is a (possibly infinite) sequence $q_0, \ell_1, q_1, \ell_2, q_2, \ldots$ such that:
$q_0 \in Q_0$, and
$\forall i \geq 0 (q_i, \ell_{i+1}, q_{i+1}) \in \delta$.

Thus, the execution includes the labels as well as the states.

The *trace* of an execution is the subsequence consisting of all the external (input and output) actions.

A labeled state machine is not very different from an ordinary state machine.
It just adds the labels.
In particular, the methods we have already studied for proving things about ordinary state machines—invariants and termination functions—can also be used for labeled state machines.

The labels will let us to other things, namely, to talk about system structure.

The rest of this section contains a few other examples of labeled state machines (and their external behavior).

2.3 Relation example

Consider a state machine that keeps track of a binary relation on a set A.
The relation starts out empty, and a user can add and remove pairs.
(Can think of adding and removing edges from a digraph.)

The user can also perform some queries:
Asking if a particular pair is (currently)
– In the relation
– In the reflexive transitive closure of the relation (that is, connected by some walk
And the state machine is supposed to respond appropriately.

This interaction can go on forever.

Labeled state machine:
The state consists of the current relation, plus some bookkeeping info about active operations.

Q:
rel, a binary relation on A
RTC, a binary relation on A
$reqtype \in \{``add'', ``remove'', ``in'', ``RTC''\} \cup \{null\}$
$reqpair \in A \times A \cup \{null\}$
$retval \in \{``OK''\} \cup Bool \cup \{null\}$

Q_0:
rel and RTC initially contain no pairs (empty relation).
All the bookkeeping components are $null$.

L:
Input labels:
$request\text{-}add(a, b)$, $a, b \in A$
$request\text{-}remove(a, b)$, $a, b \in A$
$request\text{-}in(a, b)$,...
$request\text{-}RTC(a, b)$,...

Output labels:
$return(``OK'')$
$return(b)$, $b \in Boolean$

Internal labels:
$compute$

δ:

I won't write these steps out formally.
The input steps just record the request type in the bookkeeping state components.
If there is an active request, the new request is ignored.

The output steps remove the responses and requests from the bookkeeping state components.

The compute steps do the real work.
Processing the add step involves putting the new pair into *rel*.
If it wasn't already there, then RTC is recalculated.
Do this in one big step (or in several steps, e.g., using successive doubling—this would require a counter to keep track).
Finally, set $retval :=$ "OK''.

Remove is analogous.

request-in requires just one compute step, which looks up the answer and puts it into response component.

request-RTC analogous, looking up in RTC.

Example trace:
Assume $A = \{1, 2, 3, 4\}$.
request-add$(1, 2)$, *return*("OK''),
request-add$(2, 3)$, *return*("OK''),
request-add$(3, 4)$, *return*("OK''),
request-RTC$(1, 4)$, *return*(*true*),
request-remove$(3, 4)$, *return*("OK''),
request-RTC$(1, 4)$, *return*(*false*).

What might we prove about this machine?
If an execution does not contain two consecutive input steps without an intervening output step, then:
– It does not contain two consecutive responses.
– Responses correctly reflect the preceding add and remove operations.

Of course, must state these more formally.

Again, note that these properties only talk about externally-visible behavior.

Alternative impl:
Lazier evaluation of the RTC, only when an RTC request is made.
Very different implementation, but same external behavior.

2.4 Caching example

Modify the caching service example to model interaction with two users, one at each cache.

That is, add labels for inputs from users:
request-read$(x, 1)$ is an input from user 1 requesting to read the value of x from his local cache (*cache1*).
request-read$(x, 2)$ same, for user 2.
request-write$(x, v, 1)$, *request-write*$(x, v, 2)$ are requests by the two users to write value v in location x in memory (not cache).

These steps just record the request (if there's no other active request at the same location).
Requires bookkeeping info, as usual.
Note that we allow operations to be active simultaneously at both locations.

Output labels for corresponding returns:
return-read$(v, 1)$, *return-read*$(v, 2)$
return-write$(``OK'', 1)$, *return-write*$(``OK'', 2)$.

These steps just return determined responses.

Internal labels:
perform-read does the actual read from a local cache, after it's been requested and the value is non-null.
perform-write does the actual write to memory, after it's been requested.
copy copies a value from memory to a cache.
drop drops a value from a cache.

Copy and drop can happen any time, not just during the performance of a read or write operation.

This general description applies to both weak and strong caches.
The only difference is that, in the strong case, a perform-write step cleans out that location in both caches.

Example trace for strong caching system:
Assume that the value set is N.
request-write$(x, 3, 1)$ (user 1 requests to write value 3 in location x),
request-read$(x, 2)$ (user 2 requests to read value of x from its cache),
return-read$(3, 2)$,
return-write$(``OK'', 1)$.

This is OK because it arises, e.g., from an execution where:
user 1 requests to write,
the actual write happens to memory,

the write gets copied into cache2,
user 2 requests to read,
the read happens locally,
the read returns,
the write returns.

Example trace for weak caching system:
$request\text{-}write(x, 3, 1)$ (user 1 requests to write value 3 in location x),
$return\text{-}write(``OK'', 1)$,
$request\text{-}read(x, 2)$,
$return\text{-}read(0, 2)$.

This can arise from an execution where the memory starts with 0 in location x,
and this gets copied to cache2,
then user 1 requests the write, which gets performed and returns, but without cleaning the old value out of cache2.
Then the read gets performed from cache2, getting the old value.

This trace cannot arise for strong cache, because the old value would get cleaned out when the write occurs to memory.

Things to prove?
Weak cache: Fairly weak guarantees, e.g.:
Returns either the initial value or some value someone requested to write.
(Can impose some limitations on order of values returned.)

Strong cache: Some stronger guarantees, e.g.:
(Special case): Suppose a write operation W to location x completes before a read R of x begins.
Suppose there are no active writes to x after W completes.
Suppose R returns.
Then R returns the value written by write W.

3 Abstraction

Now we have enough math machinery to describe what it means for one computational process to "implement" another.

Definition:
Suppose that A and B are two labeled state machines with the same input and output labels.
Then we say that A *implements* B provided that every trace of A is a trace of B.

This says that anything that A does is also done (or, allowed) by B.
I'm not paying any attention to liveness (eventuality) here—just correct results.

The following three examples illustrate some of the many different uses of abstraction.
GCD example: Algorithm optimization.
Relation example: Abstract data types and their implementations.
Cache example: Concurrent program implementing a global service.

3.1 GCD example

Let A be the labeled state machine I described earlier, for calculating GCD using quotient-remainder.
Let B be the machine for GCD using subtraction.

Earlier, I said they both compute the GCD.

One way to prove this is to break up the job into two pieces:

1. Showing that B correctly computes the GCD (if only one request happens, and if there is a return value, then it is the GCD).

We already did that, using invariants.

2. Showing that A implements B, that is, every trace of A is a trace of B.

How?
Well, consider any execution of A, and try to construct an execution of B having the same trace.
The important part of the execution of A involves a series of steps, each of which moves from (x, y) to a new pair, consisting of the smaller of x and y and the remainder when the larger is divided by the smaller.
Replace each such step by a series of subtraction steps, ending up with the same new pair (in one order or the other).

Example: Consider the problem $GCD(112, 84)$.
The quotient/remainder algorithm A goes to $(84, 28)$, then $(28, 0)$, then returns the 28.
Replace the first division step by one subtraction step, which yields $(28, 84)$.
Then replace the second division step by two subtraction steps, yielding $(28, 56)$, then $(28, 28)$.
Now done, answer returned by B is 28.

The idea is simply that the remainder can be found by repeated subtraction.

Formally, this gives a recursive construction of an execution of B, working step-by-step in the A execution.
Can prove by induction on number of steps in the A execution that the traces are the same.

Theorem:
The remainder-based GCD labeled state machine implements the subtraction-based GCD

labeled state machine.

3.2 Relation example

Let B be the relation labeled state machine I described earlier, with operations for adding and removing pairs, and determining if a pair is in the relation, or is in the relation's reflexive, transitive closure.
Its basic data values are relations, and the steps of the state machine involve performing high-level relational operations like taking closures.
(Or perhaps successive squaring of relations.)

Now, an actual implementation of this could be written in terms of a lower-level representation of the relation, e.g., as an adjacency matrix.

The tests would involve looking up particular elements in the matrix.

The transitive closure operation would involve successive squaring of matrices, using matrix multiplication.
(Boolean matrix multiplication, with ands and ors instead of times and plus.)

Let A be a labelled state machine of this kind.
The state would consist of matrix representations of the relation and of its RTC.
Plus some bookkeeping information to keep track of what operations are in progress.

Now, assume we already understand B, and believe that its external behavior is correct.
To convince ourselves that A's external behavior is correct, we have only to show that every trace of A is also a trace of B.

How?
Given an execution of A, try to construct an execution of B with the same trace.
Just replace the matrix ops by the corresponding relational ops they represent.
E.g., if a step squares a matrix in the A execution, then define a corresponding step in a B execution that squares the relation represented by the matrix.

Theorem:
The matrix-based labeled state machine implements the abstract relation-based labeled state machine.

Alternative impl:
Let A be a labeled state machine in which the relation and its RTC are represented by adjacency lists.
Do the operations in terms of the adjacency lists.
Again, every trace of A is a trace of B (the relation-based machine).
That is, A implements B.

3.3 Caching example

3.3.1 Strong vs. Weak Caching

We can let A be the strong caching system, and B the weak caching system.
It should be obvious that A implements B.
Why?

Start with an execution of A, construct a execution of B with the same trace.

Not hard—the execution of A is essentially doing a special case of what is allowed by B.
Just mimic the same execution in B.
The only difference:
In A, a write to memory automatically cleans out both caches.
In the B execution, this doesn't happen.
But, we can let the A perform-write step correspond to a combination of several steps in B: do the write, then clean out both caches.

Theorem:
The strong caching system implements the weak caching system.

But note that the other direction doesn't work:
Some executions of the weak caching system don't have any executions of the strong caching system with the same trace.
We saw an example above.

Theorem:
The weak caching system does not implement the strong caching system.

3.3.2 Strong caching vs. memory only

Can say something more about a strong caching system, though:

It looks, from the outside, just like a single centralized memory, without any caches.

A little more precisely, let A be the strong caching system, as before.
Let B be a "centralized memory machine":
A state machine whose state contains:
– $memory$, mapping addresses to values as before.
– Bookkeeping info to remember active operations.
A read or a write, at either location, arrives via an input step, gets performed on the central memory, and gets a response via an output step.

Theorem:
The strong caching system state machine implements the centralized memory state machine.

Proof:
Start with an execution of the strong caching system.
Construct a corresponding execution of the centralized memory state machine, step-by-step.
Inputs and outputs match up one-for-one.
When a read is performed in the caching system, A, it's done on the local cache.
However, since it's a strong caching system, the same value is in the memory.
Let the read in B get performed from the centralized memory (same location).
When a write happens in A, it happens to the central memory.
Corresponding write in B is also to the central memory.

Can prove using induction that the same external labels appear.

A key part of that proof is observing a relationship between the states of the two algorithms after any number of steps of A (and corresponding steps of B):
The memory in B is equal to the memory in A.

3.4 Abstraction relations—Definitions

Proofs that one state machine A implements another state machine B are often carried out in a very stylized way, in terms of individual steps rather than entire executions.

Like the way we proved invariants by showing something about the initial state and about each step.

Definition:
Suppose A and B are labeled state machines with the same input and output labels.
A relation R from states of A to states of B is called an *abstraction relation* from A to B provided that the following conditions hold:
(1) (Start condition) For every start state q_A of A, there exists a start state q_B of B such that $(q_A, q_B) \in R$.
(2) (Step condition) For every step (q_A, l, q'_A) of A and every q_B such that $(q_A, q_B) \in R$, there is a sequence of 0 or more steps of B that:
– starts with q_B,
– ends with some q'_B such that $(q'_A, q'_B) \in R$, and
– has the same trace as the step of A.

This gives a recipe for how to construct an execution of B for any given execution of A.
First, it shows how to get started (pick a start state of B that is related to the start state of A).
Then it shows how to construct successive steps of B to correspond to the successive steps of A.
The invariant I of A is used
This yields a constructive proof of the main result about abstraction relations:

Theorem:

If there is an abstraction relation from A to B, then A implements B.

Actually, a stronger version of this theorem is often used in practice.
This allows invariants previously proved for A and B to be assumed when showing step (2).

Prove using induction.

Example:
Show strong cache implements centralized memory.
(Sketch)

Key is to define the right relation R between the states of the two machines:
High-level memory = low-level memory.
Plus, bookkeeping corresponds (same types of ops are active in both).
With this, the two conditions are easy to show:

Start:
Start with same memory contents in both.

Step:
Uses cases, based on all the kinds of steps of A.
For each, need to say what the corresponding steps of B are (as we discussed earlier).
And must check the stated conditions.

In this example, the relation is actually a function, from the implementation state to the specification state.
That's not always the case.
E.g., for the GCD correspondence, will want to allow the two numbers to appear in either order.

This abstraction method is very important.
More details on this method, and many examples, appear in later courses: 6.170: Data abstraction.
Distributed algorithms.
POCS.

4 Composition

The rest of the story is composition:
building bigger systems out of smaller objects and services.
E.g., a complicated distributed system out of communication services, database services, fault-detectors,...

Again, the labels provide the right support.
Basically, combine labeled state machines by matching up output labels in one component with input labels in other components.

Builds a bigger labeled state machine.

We don't have time to cover this in 6.042.
Again, left for later courses, e.g. distributed algorithms.