Composing and Decomposing Computational Processes

Topics for today:

- External behavior
- Labeled state machines
- Implementation and levels of abstraction
- Abstraction Theorem
- Composition

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.

Can also include more complex systems.

Examples:
Computer operating systems
Distributed data management (storage and retrieval) systems
Communication systems

However, such systems are 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., composing relations, coloring a graph node.
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.

E.g., matrix representation for a relation, operations involving matrix elements to do the composition.
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.
Basis of object-oriented programming, component-based software design.
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?

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

For system decomposition, need a little more:

2. Need to capture the idea that some parts of a computational process (system 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.

3. Need to capture the idea that one computational process *implements* another, so it can be substituted for it in larger systems.
This will be based on saying that any externally-visible behavior of one process can also be exhibited by the other process.

4. Need to capture the idea that two computational processes can be *composed*, so that they operate in parallel and interact.
When they interact, all that each component sees about the other one is its external behavior.

The notions of external behavior, implementation, and composition are fundamental to the process of decomposing systems.

In the rest of this lecture, I will:
1. Define externally-visible behavior of a state machine (and give examples).
2. Define what it means for one state machine to "implement" another one (and give examples).
3. Give a theorem that expresses one method for showing that a state machine implements another one.

4. Define how state machines can be composed (and give an example).

2 Externally-visible behavior

2.1 Dynamic Set

Describe a state machine that maintains a "dynamic" (changing) subset of the set $S = \{1, \ldots, 100\}$ of positive natural numbers ≤ 100.
The set starts out empty, and a "client" can add and delete elements.
The client can also ask if a particular element is (currently) in the set. And the state machine is supposed to respond appropriately.
The interaction can go on forever.

In the particular state machine I have in mind, information about the set is maintained in the form of a function from S to $\{0, 1\}$, with a 0 indicating that the element is in S and 1 meaning that it isn't.
But there could be other reps, e.g., a sorted or an unsorted list of the elements.

The client doesn't care about the representation of the set.
He only cares that when he asks if an element is in the set, he gets the right answer, based on the adds and deletes that he has done previously.

So, the new issue here is distinguishing the part of the operation of the state machine that should be visible to the client (his requests and responses) from the part that is invisible, simply part of the implementation.

The key idea is to explicitly classify the transitions as visible or invisible.
Act for the visible ones, it's usually convenient to classify them as input or output transitions.

Example: Dynamic set

Let's try to define a state machine:

Q: The set of functions $S \Rightarrow \{0, 1\}$
Q_0: A singleton set of functions, consisting of just the function that maps all elements of S to 0. (This represents the empty subset of S.)

δ:
Has three kinds of transitions: input transitions, output transitions, and internal transitions:
The internal steps will do the real work; the inputs and outputs will just be for communication with the client.

Input steps:
Request that an operation be done.
What state change do they cause?
None, as far as the data we've already put into the state.
So, actually, we should include some other data in the state, to keep track of the progress of oper-

ations.

Rewrite:

Q: The set of "tuples" consisting of components:
fun, a function from S to $\{0,1\}$.
req, either $add(v)$, $delete(v)$, $lookup(v)$, or *null*, which represents no active request
resp, either OK, a Boolean b, or *null*, which indicates no ready response.

Q_0:
fun: As before, the function that maps everything to 0, representing the empty set.
req: *null*
resp: *null*

Now, back to the transitions:

δ
Input steps: The client requests an operation. Three kinds.

$request(add(v))$:
Can occur anytime
Effect: if $req = null$ then $req := add(v)$

And similarly for the other two kinds of inputs.
A technicality: If a request should arrive when another request is active, it gets ignored. Just for convenience.

Internal steps: Perform an operation internally. E.g.:

$perform(add(v))$:
Can occur if: $req = add(v)$ and $resp = null$
Effect is to set $fun(v)$ to 1 and *resp* to OK

$perform(delete(v))$
Can occur if: $req = delete(v)$ and $resp = null$
Effect: $fun(v) := 0$; $resp := OK$

$perform(lookup(v))$:
Can occur if: $req = lookup(v)$ and $resp = null$
Effect: $resp := fun(v)$

Output steps: The response goes back to the client and the state machine gets ready for another operation.

$respond(r)$:
Can occur if: $resp = r$
Effect: $req := null$, $resp := null$

How can we talk about externally-visible behavior of this machine?

Make the labels $request(o)$, $perform(o)$, $respond(r)$ part of the formal model.

Augment the basic definition of a state machine to allow "labels" for steps (see next subsection). Also, augment the definition of an execution to include the labels.

Example 2.1 An execution of the labeled state machine for dynamic set:
$(\lambda(x)(0), null, null)$;
$request(add(17))$; $(\lambda(x)(0), add(17), null)$;
$perform(add(17))$; $(\lambda(x)(\text{ if } x = 17 \text{ then } 1 \text{ else } 0), add(17), OK)$;
$respond(OK)$; $(\lambda(x)(\text{ if } x = 17 \text{ then } 1 \text{ else } 0), null, null)$;
$request(lookup(6))$; $(\lambda(x)(\text{ if } x = 17 \text{ then } 1 \text{ else } 0), lookup(6), OK)$;
$perform(lookup(6))$; $(\lambda(x)(\text{ if } x = 17 \text{ then } 1 \text{ else } 0), lookup(6), 0)$;
$respond(0)$; $(\lambda(x)(\text{ if } x = 17 \text{ then } 1 \text{ else } 0), null, null)$.

External behavior:
What should be considered to be the externally-visible behavior of this process?
Can reasonably say that what the client 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 2.2 From the execution above, we can extract the trace:
$request(add(17))$, $respond(OK)$, $request(lookup(6))$, $respond(0)$.

Notice that this trace only tells us the input and output—it says nothing about how the process actually computed the answer (by maintaining a Boolean-valued function).
The machine might have used another algorithm or representation.

What might we prove about executions of this labeled state machine?
For example:
1. That all the responses that occur are correct (give the right answer to each operation, based on the preceding operations).
2. That each request eventually gets met with a corresponding response.
Technically, this must also assume a restriction on the client—that the only request operations after any previous ones have completed.

Property 1 is a basic correctness property. We will see how to prove this in a few minutes.
Property 2 is an eventuality property, and is a little tricky:
We have to hypothesize a live execution.
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.

2.2 Labeled state machines—definitions

Motivated by what we did for the dynamic set state machine, we give general definitions for labeled state machines, and their executions and traces.

A *labeled state machine* consists of (Q, Q_0, L, δ), where:
Q, Q_0 are sets of 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 an 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 now includes the labels as well as the states.

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

A labeled state machine is not very different from an ordinary state machine.
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.

However, the labels will let us do other things, namely, to talk about external behavior, abstraction, and composition.
The rest of this lecture contains some more examples of labeled state machines and their external behavior.

Last semester's notes (spring 00) contain other examples of labeled state machines.

2.3 Caching

Modify the caching service example to model interaction with clients, one at each cache.
That is, add transitions for requests by clients to read and write and responses to those operations.

Inputs:
$\mathit{request}(\mathit{read}(a), i)$ transition is an input from client i (a client at location i) requesting to read the value in address a.
$\mathit{request}(\mathit{write}(a, v), i)$, is a request to write value v in location a in memory (not cache).
These steps just record the requests.
A new twist: Unlike the previous machine, we will allow operations to be active simultaneously at separate locations.

Outputs:
$\mathit{respond}(v, i)$, which returns a value at location i, in response to a read request at that location.
$\mathit{respond}(OK, i)$, response to a write.

Internal steps:
$perform(read(a), i)$ does the actual read, reading from the local cache at location i.
$perform(write(a, v), i)$ does the actual write, writing to the main memory.
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 the address in all the caches.
(Actually, many algorithms would make an exception and not clean out the cache at the location at which the write is being submitted—instead, that cache would be written with the new value, along with the main memory.)
More detailed model for weak caching system:

Q: *mem*, *cache*, same types as before
req, a mapping from I (locations) to the set of values $\{read(a), write(a, v), null\}$
resp, a mapping from I to the set of values V (the value set) $\cup \{OK\, null\}$

Q_0: *mem* everywhere v_0, *cache* everywhere *null*, bookkeeping components all *null*

L: Input:
$request(read(a), i)$, $a \in A$, $i \in I$
$request(write(a, v), i)$, $a \in A$, $v \in V$, $i \in I$

Internal:
$perform(read(a), i)$, $a \in A$, $i \in I$
$perform(write(a, v), i)$, $a \in A$, $v \in V$, $i \in I$
$copy(a, i)$, $a \in A$, $i \in I$
$drop(a, i)$, $a \in A$, $i \in I$

Output:
$respond(v, i)$, $v \in V$, $i \in I$
$respond(OK, i)$, $i \in I$

δ: Input:
$request(read(a), i)$
Can occur: anytime
Effect: if $req(i) = null$ then $req(i) := read(a)$

$request(write(a, v), i)$
Can occur: anytime
Effect: if $req(i) = null$ then $req(i) := write(a, v)$

$perform(read(a), i)$, $a \in A$, $i \in I$
Can occur: $req(i) = read(a)$, $cache(a, i) \neq null$, and $resp(i) = null$
Effect: $resp(i) := cache(a, i)$

$perform(write(a, v), i)$
Can occur: $req(i) = write(a, v)$ and $resp(i) = null$
Effect: $mem(a) := v$; $resp(i) := OK$

$copy(a, i)$
Can occur: anytime
Effect: $cache(a, i) := mem(a)$

$drop(a, i)$
Can occur: anytime
Effect: $cache(a, i) := null$

$respond(r, i)$
Can occur: $resp(i) = r$
Effect: $req(i) := null$; $resp(i) := null$

For strong caching, same except for the $perform(write)$:

$perform(write(a, v), i)$
Can occur: $req(i) = write(a, v)$ and $resp(i) = null$
Effect: $mem(a) := v$; for all i, $cache(a, i) := null$; $resp(i) := OK$

Example 2.3 Trace for strong caching system:
Assume that the value set V is N, with $v_0 = 0$, and $I = \{1, 2\}$.
$request(write(a, 3), 1)$ (client 1 requests to write value 3 in address a),
$request(read(a), 2)$ (client 2 requests to read value of a),
$respond(3, 2)$,
$respond(OK, 1)$.

This is OK because it arises, e.g., from an execution consisting of steps with the following labels:
$request(write(a, 3), 1)$
$perform(write(a, 3), 1)$
$copy(a, 2)$, the new value gets copied into cache2,
$request(read(a), 2)$
$perform(read(a), 2)$, the read happens locally, from cache 2,
$respond(3, 2)$,
$respond(OK, 1)$.

Example trace for weak caching system, but not strong caching system:
$request(write(a, 3), 1)$
$respond(OK, 1)$.
$request(read(a), 2)$
$respond(0, 2)$,

This can arise from an execution where the initial value of 0 in memory address a gets copied to cache 2,

then client 1 requests the write, which gets performed and returns, but without cleaning the old value out of cache 2.
Then the read gets performed from cache 2, 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.
(Could show some weak constraints on order of values returned.)

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

3 Implementation

3.1 Definition

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

Definition 3.1 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 can do, B can also do.
I'm not paying any attention to liveness (eventuality) here—just correct results.

This notion of implementation is used to represent several different notions, including:
Algorithm optimization. A might be an optimized (more efficient) version of B.

Abstract data types and their implementations. See dynamic set example below.
Concurrent program implementing a global service. See cache example below.

3.2 Dynamic set

This example illustrates an implementation relationship between an abstract data object and a more detailed, "lower level" implementation.

Recall the dynamic set example, which maintains a subset of the set $S = \{1, 2, \ldots, 100\}$.

Let A be the state machine I described earlier today, with operations for adding and deleting elements and testing if an element is currently in the set.
The implementation works in terms of a Boolean-valued function.
Alternative implementations are possible.
We would like to know that it implements an actual dynamic set.

Now suppose I express what I mean by a dynamic set using a more abstract, "higher-level" state machine B.
I'll only worry about correct answers, no liveness.

Define B:
It will be similar to A, but instead of talking about the function *fun*, I'll talk about the actual set *dynamic-set* being maintained:

Q: *dynamic-set*, a subset of S
req, *resp*, as before

Q_0: *dynamic-set*: $\emptyset$
req, *resp*: *null*

δ: Inputs: *request*(*add*(v)), etc. As before.

Internal steps:
perform(*add*(v)):
Can occur if: $req = add(v)$ and $resp = null$
Effect: $dynamic\text{-}set := dynamic\text{-}set \cup \{v\}$; $resp := OK$

perform(*delete*(v))
Can occur if: $req = delete(v)$ and $resp = null$
Effect: $dynamic\text{-}set := dynamic\text{-}set - \{v\}$; $resp := OK$

perform(*lookup*(v)):
Can occur if: $req = lookup(v)$ and $resp = null$
Effect: if $v \in dynamic\text{-}set$ then $resp := 1$ else $resp := 0$

Outputs: As before.

It should be clear that this machine represents what most of us would consider to be a dynamic set.
The set operations are performed directly.
In fact, we can take B as a *definition* of what we mean by a dynamic set.
More precisely, the traces of B are the ones that we will allow a dynamic set implementation to exhibit.

Now, if we want to show that A is a correct implementation of a dynamic set, it would be enough to show that any trace of A is also a trace of B.
So, consider any execution of A and try to construct an execution of B having the same trace.

How? Starting with the fixed execution of A, construct an execution of B that "mimics" it, step-by-step.
The input and output steps do the same thing in both algorithms.
The interesting steps are the $perform(read)$ and $perform(write)$ steps.
Here, replace the function operations by the corresponding set operations.
E.g., where the execution of A sets $fun(v) := 1$, replace by a step that explicitly adds v to $dynamic\text{-}set$.
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 3.2 The function-based dynamic set state machine A implements the explicit dynamic set state machine B.

The implication of this is that one could write a program using the abstract dynamic set object.
Then later replace it with the function-based implementation.
And the program should still work correctly.

Alternative implementations:
1. List (sequence) of elements of the set, any order, no repeats.
Add checks to see if the element is already in the list, and if not, adds it at the end.
Delete searches for the first occurrence and deletes.
Lookup searches for the first occurrence.

2. List of elements of the set, any order, allowing repeats.
Add just adds at end.
Delete searches for and removes all occurrences.
Lookup searches for first occurrence.

3. Sorted list, no repeats
Add, delete, lookup all search for the right place using binary search.

All three of these talk about lists (sequences). Can "refine" further, representing the abstract lists by linked lists, arrays, or trees (consider binary search trees, as I mentioned briefly in the lecture on trees, as a way of implementing the sorted list).
Can show these more "refined" versions implement the various list-based algorithms, which in turn implement the abstract dynamic set.
Can use transitivity (of set inclusion, for sets of traces) to show that the lowest-level implementations actually implement the highest-level dynamic sets.

"Successive refinement".

3.3 Caching example

3.3.1 Strong vs. Weak Caching

Now 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.

Easy—the execution of A is essentially doing a special case of what is allowed by B.
So use (essentially) 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: $perform(write)$, $drop$, $drop$, etc.

Theorem 3.3 The strong caching system implements the weak caching system.

The other direction doesn't work, however. 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.
Thus, the weak caching system does not implement the strong caching system.

3.3.2 Strong caching vs. centralized memory

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: just mem, mapping addresses to values as before, plus the standard bookkeeping information for active operations.
A read or a write, at any location, arrives via an input step, gets performed on the central memory, and gets a response via an output step:

Q: mem, req, $resp$, same types as for SC
Q_0: As for SC.
L: Input, Output: As for SC.
Internal:
$perform(read(a), i)$, $a \in A$, $i \in I$
$perform(write(a, v), i)$, $a \in A$, $v \in V$, $i \in I$
No $copy$ or $drop$.

Transitions:
Input, Output: As for SC.

Internal:
$perform(read(a), i)$
Can occur: $req(i) = read(a)$ and $resp(i) = null$
Effect: $resp(i) := mem(a)$

$perform(write(a, v), i)$
Can occur: $req(i) = write(a, v)$ and $resp(i) = null$
Effect: $mem(a) := v$; $resp(i) := OK$

Theorem 3.4 The strong caching system state machine SC implements the centralized memory state machine CM.

Proof. Start with an execution of SC, and construct a corresponding execution of CM, step-by-step.
Inputs and outputs match up one-for-one.

When a read is performed in the caching system, SC, it's done on the local cache.
However, since it's a strong caching system, the same value is in the memory.
The corresponding read in CM is performed from the memory.

When a write is performed in SC, it is performed on mem.
Corresponding write in CM is also to mem.

When copy, drop occur in SC, no steps correspond in CM.

Can prove using induction on the number of steps in the execution of SC that the same external labels appear, which means that the two executions look the same externally.

A key part of that inductive proof is observing a relationship between the states of the two algorithms after any number of steps of SC and the corresponding steps of CM:
mem in CM is equal to mem in SC (and the bookkeeping corresponds).
Think of this as an invariant involving the states of the two state machines. ■

4 Abstraction relations and the Abstraction Theorem

4.1 Overview

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.

In this section, I'll give you the basic definition and theorem, and revisit our examples briefly.
That's about all the time we will have in this course to devote to these notions.
You will have a chance to see versions of them later, in 6.170 and perhaps in some systems courses or distributed algorithms.

4.2 Definition and theorem

This definition describes a correspondence (formally, a relation) between the states of A and B that is preserved while the two machines execute.

Definition 4.1 Suppose A and B are labeled state machines with the same input and output labels.

Then 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)
   If q_A is a start state of A, then there exists a start state q_B of B such that $(q_A, q_B) \in R$.
2. (Step condition)
   If q_A and q_B are reachable states of A and B, respectively, $(q_A, q_B) \in R$, and (q_A, ℓ, q'_A) is a step of A, then there is a sequence of (0 or more) steps of B that:
   – Starts with q_B,
   – Ends with some state q'_B such that $(q'_A, q'_B) \in R$, and
   – Has the same trace as the step of A (that is, it looks the same externally).

The importance of this slightly complicated-looking definition is that having an abstraction relation from A to B implies that A implements B, in the sense that every trace of A is also a trace of B:

Theorem 4.2 If there is an abstraction relation from A to B, then A implements B.

Proof. The two conditions in the definition of an abstraction function provides a method of constructing 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.

Construct this execution of B recursively on the structure of the execution of A.
Then prove by structural induction that the traces of any prefix of the execution of A and of the corresponding part of the execution of B are the same.
Follows that the traces for the entire executions are the same. ■

Note: The fact that we assume reachable states in Condition 2 means that we can use any invariants we have previously proved for A and B in showing that the step correspondence holds.

4.3 Examples

The two implementation examples I presented in the previous section can be proved correct more formally using abstraction relations. For instance:

Example 4.3 Show strong cache SC implements centralized memory CM.

The key is to define the right relation R between the states of the two machines:
High-level memory = low-level memory.
Bookkeeping corresponds (same types of ops are active in both).
That is:

$$CM.mem = SC.mem$$
$$CM.req = SC.req$$
$$CM.resp = SC.resp$$

With this, the two conditions are easy to show:

1. Start condition:
The initial states of both machines are unique.
Both *mems* are v_0 everywhere (and the *reqs*, etc. are *null*.

2. Step condition:
Proof by cases, based on all the kinds of steps of SC.
For each kind, need to say what the corresponding steps of CM are (as we discussed earlier).
And must check the stated conditions.

Key steps: The internal ones—the inputs and outputs correspond in a straightforward way.

perform(*read*): Corresponds to the same-label step in CM.
The key argument is that the same value is written into the *resp* state components in both the step of SC and the corresponding step of CM.
This is needed in order to show that the states correspond after the step.

This is not completely obvious, because in SC, the value comes from the cache, whereas in CM, it comes from the memory.

However, the memory in CM is the same as that in SC because of the assumed correspondence before the step.
And the cache value is the same as the memory value in SC because of an invariant we proved earlier for SC.
So that means the values that are obtained in SC and CM are equal.

perform(*write*): Corresponds to the same-label step in CM.
The key argument here is that, assuming a is the address being written, the new value of $mem(a)$ in CM is the same as that in SC.
Again, that is needed in order to show that the states correspond after the step.
That is pretty obvious, though, because both steps modify $mem(a)$ directly.

copy, *drop*: Correspond to no steps happening in CM.
The key argument here is that the new state in SC still corresponds to the same state of CM.
But that's easy to see, because neither of these steps change the *mem* values in SC (or bookkeeping values), but only the cache values.

In this example, the relation is actually a function, from the implementation state to the specification state.
That's not always the case.
Common situation where it isn't: when A represents an optimized version of B.
This abstraction method is a very important way of thinking about systems.
More about this method, and many examples, appear in later courses. And this way of thinking will also be useful elsewhere (courses, real design).

5 Composition

Composition is an operation for combining labeled state machines.
It is used to build models of bigger systems out of models for its components.
E.g., a model for a complicated distributed system out of models for communication services, database services, fault-detectors,...
The composition operator allows us to understand the behavior of a complex system, once we understand the behavior of the separate components.

The composition operation combines labeled state machines by "matching up" external labels of different machines, usually output labels of some machines with input labels of other machines. The executions of the bigger machine describe what would happen if the different modeled components actually operated together.

Example 5.1 Consider a version *Split-SC* of *SC* in which the cache removals happen in separate steps. That is, a write now consists of a request, then a write to $mem(a)$, then one-by-one, removals from each $cache(a,i)$, then finally, a response.

It is not hard to see that this does not, by itself, implement *CM*:
Example execution: Reads out-of-order
Write starts, writes to memory, throws away some of the cache values; Some copies occur, putting the new value into some of the caches; Read1 starts, gets the new value from its cache, returns; Read2 starts, gets the old value (still in its cache), returns; Write continues, throwing away the rest of the cache values. This order gives something different from what could be observed when interacting with *CM*.

However, *Split-SC* would work correctly in certain contexts, e.g., in an environment in which no two operations on the same memory location a ever overlapped.
This kind of strategy is often used in practice, and is implemented by having the client processes that request these operations engage in a preliminary protocol, involving securing "locks" on data items before accessing them.

Can represent the situation by modeling the environment as a labeled state machine *Env*, whose outputs are the requests and inputs the responses, for *Split-SC*. *Env* just keeps track of which locations have currently active operations for which memory addresses.

Q: $locks$, a mapping from A to subsets of I,
indicating for each memory address, which locations have currently active operations
Q_0: for all a, $locks(a) = \emptyset$
L: Input and output reverse those of *Split-SC*:

- Input:
  - $respond(v,i)$, $v \in V$, $i \in I$
  - $respond(OK,i)$, $i \in I$
- Output:
  - $request(read(a),i)$, $a \in A$, $i \in I$
  - $request(write(a,v),i)$, $a \in A$, $v \in V$, $i \in I$
- Internal: None

Transitions:

$respond(*, i)$
Can occur: anytime
Effect: for all a, $locks(a) := locks(a) - \{i\}$
(That is, throw away locks held by i.)

$request(read(a), i)$
Can occur: $locks(a) = \emptyset$
Effect: $locks(a) := \{i\}$
$request(write(a, v), i)$
Can occur: $locks(a) = \emptyset$
Effect: $locks(a) := \{i\}$

The system of interest is really the combination of *Split-SC* and the environment *Env*, interacting in such a way that the requests that are made by *Env* are identified (synchronized) with the requests that are received by *Split-SC*, and similarly for responses.
This combination is expressed formally by the composition of the two machines.

I won't give the formal definition of the composition, but will indicate how it comes out in this example:

The state set of the composition is the Cartesian product of the state sets of the two component state machines.
That is, it has state variables: *mem*, *cache*, *req*, *resp*, and *locks*.

Transitions correspond to all the transitions of the individual machines, combining those with the same labels.
For instance, the transitions for $request(read)$ result from combining the conditions and effects for that label in the two machines:

$request(read(a), i)$
Can occur: $locks(a) = \emptyset$
Effect: $locks(a) := \{i\}$; if $req(i) = null$ then $req(i) := read(a)$

Now, what might we want to prove about this combined state machine?
Well, the intuition that motivated the definition of *Env* was that, with its restrictions, the *Split-SC* memory would behave like *CM*.
Formally, we could take the composition A of *Env* and *Split-SC*, and show that it implements the composition B of *Env* and *CM*, in the sense that every trace of A is also a trace of B.
(The external actions we're considering for these traces are just the same ones as before—the requests and responses.) That is, when *Env* interacts with *Split-SC*, *Env* cannot see any traces that it couldn't see when interacting with *CM*.

Again, this correspondence can be proved using the Abstraction Theorem and invariants. LTTR.

That's all I have time to show here about composition. More on this in later courses, e.g. distributed algorithms.