[1] This lecture is all about concurrent composition. We have been using [2] sequential composition, P dot Q. In a sequential composition, [3] P and Q must have exactly the same state space. That's because the final state of P is the initial state of Q. [4] Concurrent composition, P parallel Q, means executing P and Q at the same time. When we put 2 or more programs in parallel, we call them processes. For concurrent composition, [5] P and Q must have completely different, disjoint state spaces. That's so that they won't interfere with each other when they are being executed. The variables of the composition are the variables of all the processes being composed.

[6] If we aren't considering time and space, then concurrent composition is just conjunction. Behave according to specification P, and also according to specification Q. Let's see an [7] example. Suppose we have 3 variables x, y, and z, and we put x gets x plus 1 in parallel with y gets y plus 2. We have to [8] partition the variables, and obviously we have to [9] put x in the left part, and y in the right part. We could put z in either part, it doesn't matter which. I'll pick the right part for no good reason. Then [10] x gets x plus 1 is just x prime equals x plus 1, because x is the only variable in the left process. y gets y plus 2 becomes y prime equals y plus 2 and z prime equals z. And then [11] concurrent composition is just conjunction.

When you refine a specification using concurrent composition, you have to decide how to partition the variables. It's the person who introduces the concurrent composition who decides how to partition. But suppose you are presented with a program that includes a concurrent composition, and the person who wrote it failed to record the partitioning. Then you have to make a [12] reasonable guess about what partitioning was intended. Here is a way to make a reasonable guess. If either x prime or x colon equals appears in one process specification and not in the other one, then x belongs to that process. If neither x prime nor x colon equals appears at all, then x can be placed on either side of the partition. For [13] example, in variables x, y, and z again, x gets y, in parallel with y gets x. On the left we have x gets something, so [14] x belongs to the left process. On the right we have y gets something, so y belongs to the right process. On the right, we see an x, but we don't see x prime or x gets something. So on the right, x is not a variable. It's a constant. Similarly y is a constant on the left. Each process can see the initial values of all the variables, but it can change only its own variables. We don't see z prime or z gets anything, so we can put z on either side. And we get [15] this. x and y swap values, and they seem to do it without any temporary variable. It's a neat way to say swap, but actually the [16] implementation of a process has to make a private copy of any variable belonging to another process that it uses, and that the other process changes. So the temporary variable is supplied by the implementation.

[17] Here's another example. On the left we have b gets x equals x. On the right, x gets x plus 1. b belongs to the left process, and x belongs to the right process. We can use [18] the reflexive law of equality to replace x equals x by true. The point of this example is that you might worry that in between the two evaluations of x on the left side, the increase to x on the right side might happen, so you would get false as an answer. But that doesn't happen. On the left side, x is not a variable. If we can't even trust that x equals x, then we can say good bye to all of mathematics.

In [19] this example, on the left side, x is increased and then decreased again. So that's the same as [20] ok. And doing nothing in parallel with y gets x is the same as just [21] y gets x. Y is assigned the initial value of x. The point of the example is that you might worry that, by bad luck of timing, y gets the intermediate value of x, between the two assignments to x. But that doesn't happen. An intermediate value is local to a sequential composition. Exists x double prime. So it cannot be seen outside the sequential composition.

Here's a problem. [22] Suppose the left process is a couple of assignments to x, and the right process is a couple of assignments to y. But suppose you want the [23] second assignments in each process to use the updated values of both x and y. Each process can use the updated value of its own variable, but it can't see the updated value of the other process's variable. The solution you find in many textbooks is to use shared variables, so every process can see the current values of all the variables, and also, according to operating system textbooks, you need synchronization [24] here. You have to make sure that the second assignment in each process waits until the first assignment of the other process is finished. But the truth is, you don't need shared memory, and you don't need synchronization. You just need to write what you meant. [25] Here's what you should have written. First there's a parallel composition of 2 assignments. And then, sequentially after that, there's another parallel composition. The need for shared memory and synchronization within processes was just a symptom of writing the wrong program.

[26] I have been saying that concurrent composition is just conjunction, but that's because I wasn't talking about time. If you include time, [27] here's concurrent composition. It's still [28] P [29] and [30] Q, but [31] P might say one thing about the final time, and Q might say something contradictory about the final time. And really, the final time is the maximum of what P says it is and what Q says it is. So that's what this definition says.

[32] There are some laws about concurrent composition that might come in handy. [33] The first one is the substitution law. When you have a concurrent composition of assignments, followed by any specification, the result is the same as the specification but make substitutions for all the assignments concurrently. The law shown here is for 2 assignments, but it could be any number. The substitution law we have been using is just the special case when it happens to be one assignment. The [34] next law says concurrent composition is symmetric, and the [35] next one says it's associative, so if we have 3 or more processes, we don't need to bother with the parentheses. And the [36] first distributive law also works for both sequential and concurrent composition. It says if you have P in parallel with either Q or R, then either you have P in parallel with Q or you have P in parallel with R. The [37] other 2 distributive laws don't work for sequential composition, just for concurrent composition.

[38] Concurrent composition requires us to partition the variables, so each process can work on its own variables independently. And that's fine when the variables are all small ones, like integer variables and binary variables. But a list variable can be a lot of memory, and sometimes we need to partition within the list to allow concurrent processes to work on different parts of the list. [39] Here's the definition of array element assignment, saying that one item is changing value, and all other items are staying unchanged, and all other variables are also unchanged. But now, if this assignment occurs in a process in a concurrent composition, then it should not say that everything else is unchanged. It should just say [40] that all other items in this part of the partition are unchanged, and all other variables in this part of the partition are unchanged. As a good example [41], I'll show you how to find the maximum value in a nonempty list in log time. I [42] define specification findmax of two variables i and j to say that the maximum value in the nonempty segment of the list from i to j is put into L i. So findmax says find the maximum item in a segment, and put it at the left end of the segment. We want the maximum of the whole list, so that's [43] findmax from 0 to the length of the list, and we'll look for the answer at L 0. And [44] here's the refinement. [45] If j minus i equals 1, then it's a segment of size 1, so the maximum is the one and only item, and it's already at the left end of the segment, so there's nothing to do. [46] Else. Else means it's not 1 item, and we know it's nonempty, so it's 2 or more items, and we can split it into 2 nonempty halves, and find the maximum of each half in parallel, and put those 2 maximums at the left end of each half. Now we want the maximum of the whole segment, [47] so we take the maximum of the 2 items, and put it in the left end. And we're done. For recursive time, we need t gets t plus 1 at the start of the else-part. [48] Recursive time is the ceiling of the log of the segment length.