[1] This video is about concurrency as a programming language feature. I want to tell you about my experiences with it, and how that has shaped my opinions.

[2] It seems to me that the treatments of concurrency fall into two camps. One treats concurrency as conjunction, and the other treats concurrency as an interleaving of atomic actions. People in the conjunction camp sometimes call their version [3] “true” concurrency, because it describes activities that really do happen at the same time. So I suppose the other camp has to be called [4] “false” concurrency, because it really describes a sequence of activities. But I don't mean to be pejorative. At my slow human speed of perception, the activities on my computer do appear to be concurrent. And it's the only possible strategy when you have to implement concurrency on a [5] single processor. “True” concurrency requires [6] as many processors as there are processes. But these are just the two extreme points on a range of points. In the middle, you have more than one processor, but not as many as you have processes. So you have to be able to schedule the processors you have in a sensible way.

One issue is to decide what are the [7] atomic actions, the units that get scheduled. And then some serious gymnastics are required to achieve anything like compositionality. Another issue is called [8] “fairness”, which means scheduling the atomic actions in a fair way. I have never liked the definition of fairness that uses the words “infinitely often”, or “always eventually” because it makes fairness unobservable. To me it would make more sense to make fairness comparative, so that one schedule is more fair or less fair than another schedule. Or quantitative, so that a schedule has a fairness rating. These are issues of interleaving; they are not issues of concurrency as conjunction.

It seems to me that interleaving is really a way of [9] implementing concurrency when there are too few processors, rather than a way of defining concurrency. Concurrency must be defined [10] for all specifications, not just programs, in order to be able to [11] program by refinement.

[12] In my theory of programming (a theory formerly known as “predicative programming”, now some people call it UTP), [13] we do not specify programs; we specify computation, or computer behavior. [14] The free variables of the specification represent whatever we wish to observe about a computation. [15] Observing a computation provides values for those variables. [16] When you put the observed values into the specification, there are two possible outcomes: either the computation satisfies the specification, or it doesn't. [17] So a specification is a binary expression. If you write anything other than a binary expression as a specification, such as a pair of predicates, or a predicate transformer, you must say what it means for a computation to satisfy a specification, and to do that formally you must write a binary expression anyway.

[18] A program is an implemented specification. It is a specification of computer behavior that you can give to a computer and get the specified behavior. I also refer to any statement in a program, or any sequence or structure of statements, as a program. Since a program is a specification, and a specification is a binary expression, therefore a program is a binary expression. [19] For example, if the program variables are x and y , then the program x gets x plus one is the binary expression x prime equals x plus one and y prime equals y, where unprimed variables represent the values of the program variables before execution of the assignment, and primed variables represent the values of the program variables after execution of the assignment. [20] So you can connect specifications with any binary operators, even when one or both of the specifications are programs. [21] The is-implied-by operator is also refinement, or implementation. [22] [23] Sequential composition looks like this. Mainly it's [24] P and Q, but

the final state of P is identified with the initial state of Q. For programming by refinement, it is essential that programming constructs apply to all specifications, not just programs. [25] For example, here are two specifications sequentially composed. It's also essential that programming constructs are monotonic so we can [26] refine specifications separately, and know that the [27] composition of the implementations will refine the composition of the specifications.

The same thing has to be true for [28] parallel composition. We have to be able to compose specifications. We have to be able to [29] refine them separately, and be guaranteed that [30] the composition of the implementations will refine the composition of the specifications. [31] [32] Here is the definition of concurrency as conjunction. If I were not including time, then [33] it would be just P and Q. But the time [34] of the composition is the maximum of the times of the processes.

This definition gives us [35] all the laws that we expect for concurrency. It's symmetric, associative, and distributes in the right ways. And monotonicity is essential for programming by refinement.

A consequence [36] of the definition of sequential composition is the Substitution Law, which is a generalization of the proof rule for assignment in Hoare Logic. It says an assignment followed by any specification is that specification with a substitution. And this law is just a special case of [37] a more general law saying that a parallel composition of assignments followed by any specification is that specification with concurrent substitutions.

[38] Let's suppose we have two variables, x and y, and let's ignore time. Then x gets 2 in parallel with x gets 3 is [39] this, which is [40] false, and that's unimplementable, and that's reasonable, because the final value of x cannot be both 2 and 3.

Here [41] is a parallel composition of assignments to different variables. [42] The assignment to x says what the final value of x is, and it also says that y is unchanged. And the assignment to y says that x is unchanged. And together we get [43] false, which is not what we want. That's why we have to partition the variables, and say that [44] the left side determines x but not y, and the right side determines y but not x.

[45] Here we have x gets y in parallel with y gets x. According to the definitions I've presented, that says x and y swap values. It's a good example of the fact that when a process uses the initial value of a variable that's not in its part of the partition, it has to make a private copy.

[46] In this next example, x is in the left part of the partition, and y is in the right part. The left part makes two assignments to x, and in both assignments, [47] y refers to the initial value of variable y. Likewise in the two assignments to y on the right, [48] both occurrences of x refer to the initial value of variable x. But now, suppose we want the [49] second occurrence of y on the left to refer to the updated value after the first assignment on the right. And likewise we want the [50] second occurrence of x on the right to refer to the updated value after the first assignment on the left. The usual solution is to treat the variables as shared variables, so each process can see the current value of the variables being changed by the other process. Then you need to [51] synchronize the concurrency here, to make sure that the second assignment in each process waits until the first assignment in the other process is finished. But really, we don't need the headaches of shared variables, and we don't need new synchronization primitives. All we need to do is say what we meant. [52] We meant that the first assignment to x and the first assignment to y are in parallel, and then, sequentially following that, the second assignments to x and y are in parallel. The need for shared memory and synchronization was just a symptom of writing the wrong program.

[53] You can go a long way with this kind of concurrency. Here's an example: finding the maximum item in a list. Specification findmax, with parameters i and j, says that we find the maximum item of a nonempty segment of list L from index i to index j. The time is the log of the length of the segment. The specification is [54] refined like this. Find the maximum in the first half of the segment, and in parallel, find the maximum in the second half. Then take the maximum of those two maximums. This refinement is a reverse implication whose proof is the proof of correctness, and proof of the time bound, and it is also the program.

This simple kind of concurrency is good for a lot of programming, but there are times when you need more interaction than it offers. So we add communication channels between processes. I don't have time now to talk about communication, [55] but it's in my book; Chapter 8 is concurrency, and Chapter 9 is communication, and it's free online if you want to read it.

I first started defining concurrency as conjunction in about 1982, and I stuck with it until about 1990. Then, together with my graduate student Theo Norvell, we had what we thought was a better idea, namely concurrency with merge. I'll talk about it in a moment. So in the first edition of this book, in 1993, concurrency is a merge. When Hoare and He wrote their book on Unifying Theories of Programming, which came out in 1998, they adopted my predicative programming, and concurrency with merge. But by then, I had 8 years programming experience with it, and I realized that it was not better than the simpler concurrency as conjunction. I remember Leslie Lamport sitting in my office, telling me to go back to the simpler conjunction, and Leslie is always right. So for the 2002 edition, right up to the present edition of my book, concurrency is conjunction.

[56] What is concurrency with merge? Here's how it works. You have to make as many copies of the state space as you have processes, so that each process can work independently on its own copy. Then when the last process is finished, the results of all processes are merged together. There are lots of ways to do this; Hoare and He talk about ten of them. [57] Here's the one I used in the first edition of my book. For each variable, if all processes leave it unchanged, then its final value is unchanged. If just one process changes it, then its final value is that changed value. If more than one process changes it, then its final value is arbitrary. And [58] here's how that looks formally. It's still [59] P and Q, and the time is still the maximum of the individual process times, and the rest is talking about how to merge the results. As a variation on this, [60] instead of saying that if two processes change x then its final value is arbitrary, we could say [61] that if all processes changing x agree on its final value, then that is its final value. That's a stronger definition. We could strengthen the definition further, [62] saying that the processes don't have to agree; the final value of x is any one of the changed values. There are lots of options.

What's so good about concurrency with merge? [63] The attraction, at least for me, is that you don't have to partition the state space. That's why I put it in the the first edition of my book. But with a few years experience in using it, I found that in practice you do have to partition the state space. And it's a lot more complicated, both for the implementation, and for programming. So I went back to simple conjunction.

[64] In conclusion, it's best to keep it simple. Concurrency is just conjunction, after partitioning the state space. Interleaving is not a general definition of concurrency; it's the implementation when there are too few processors.