[1] There are two main ways for parallel processes to interact. One way is by communication channels, and we'll get to that later. The other way is by [2] sharing variables, and that's what I want to talk about now. A shared variable is one that [3] can be read and written by any process, so one process can write, that is assign a value, to it, and another process read what the first one wrote, in other words, it can use the variable. Shared variables are [4] difficult to implement because conflicting writes by different processes at the same time have to be resolved somehow. And they are [5] difficult to reason about because of those same conflicts and timing issues. An [6] interactive variable is one that can be read by any process but written by only one process. That eliminates the conflicts, and makes them [7] easier to implement and reason about. But it provides less interaction because it's just one way communication, from the writing process to the other processes. If each process has an interactive variable, then you not only get communication from all to all, just like a shared variable, but each process can be selective about where it receives information from. So there's really no loss in interaction, and there's a gain in control. [8] Boundary variables provide the least interaction. They can be read and written by only one process. But their initial value can be seen by all processes, so that's a tiny bit of interaction. They are the [9] easiest to implement and reason about. Boundary variables are the only kind we have had so far.

[10] A declaration for a new local boundary variable looks like this, where T is the type and S is the scope. In the usual binary expression notation it is [11] this. Var *a* means exists *a* and *a* prime. It's called a boundary variable because it tells us the initial value and the final value, at the boundaries of the computation, and not anywhere in between. [12] A declaration for an interactive variable looks like this, and in the usual binary expression notation it is [13] this. x is a function from time to some type. [14] The value of variable x at time t is x of t. x is a function of time, [15] but sometimes we just write x when we mean x of t, or we write x prime when we mean x of t prime, the value of x at the final time, and similarly for any other number of primes. For example, [16] we write the assignment *a* gets *a* plus x, where *a* is a boundary variable and x is an interactive variable, and what we mean is *a* gets *a* plus x of t. With interactive variables, [17] most of the laws of programming still work, but unfortunately not the substitution law.

[18] Here's what happens to programming theory when we add interactive variables. Suppose we have a couple of boundary variables *a* and b, and a couple of interactive variables x and y, and the time variable t. [19] ok means *a* prime equals *a*, and b prime equals b, and t prime equals t. You don't have to say [20] x prime equals x, which means x of t prime equals x of t, because t prime equals t. so of course x of t prime equals x of t. [21] ok just talks about the boundary variables and time. Likewise [22] assignment to a boundary variable just talks about the boundary variables and time.

[23] Assignment to an interactive variable is a bit different. First of all, it has to take [24] nonzero time. It can't be instant because if t prime equals t, then x of t prime equals x of t. So it has to take time. [25] *a* prime equals *a* and b prime equals b, as usual for boundary variables. x prime equals e, which looks as usual, but it means x of t prime equals e. At the end of the assignment, x has the right value, but it doesn't say anything about the value of variable x during the assignment. That's important, because it means that another process, looking at the value of variable x, better not look during an assignment to x, because its value is unknown during the assignment. And this last part says that the value of interactive variable y is unchanged during the assignment. It means y of t double prime equals y of t, for all t double prime between t and t prime. It's a bit complicated, and that's why the substitution law doesn't work. [26] Sequential composition is as usual, with just the boundary variables and time needing quantification. [27] Concurrent composition starts off as usual, and then there's an extra part to say that the values of interactive variables belonging to the faster process don't change value while the slower process is finishing.

[28] Let's see an example. We can stick with boundary variables *a* and b, interactive variables x and y, and extended natural time variable t. The example has two processes in a concurrent composition, so we have to partition the variables. [29] Obviously x goes with the left process, and y goes with the right process. *a* could go with either process, so I'll put it in the left process. And b could go with either process, and for no good reason I'll put it in the right process. With that partitioning, [30] x gets 2 is *a* prime equals a and x prime equals 2 and I have to choose a nonzero time so I'll say time 1, for this and each of the assignments. I don't have to talk about b and y because they are not variables in this process. [31] The next assignment says *a* is unchanged, x prime equals x plus y, and t prime equals t plus 1. And the [32] next one is identical. And they are in parallel with [33] b prime equals b and y prime equals 3 and t prime equals t plus 1, followed by [34] b prime equals b and y prime equals x plus y and t prime equals t plus 1. Next I want to deal with the sequential compositions. I'll skip the details, and perhaps you will just believe me that [35] this is the result. In the left process, or I guess I mean the top process, *a* is unchanged, and x at t plus 1 is 2, and x at t plus 2 is the sum of x and y at t plus 1, and x at t plus 3 is the sum of x and y at t plus 2 and the time taken is 3. In the bottom process b is unchanged, and y at t plus 1 is 3, and y at t plus 2 is the sum of x and y at t plus 1, and the time taken is 2. Now for the concurrent composition [36]. It's just conjunction, except that the time is the maximum of the times for the two processes, and for the shorter process, which is the bottom one, we have to say that its interactive variable y stays unchanged for the time while the slower process is finishing, so that's [37] y at t plus 3 equals y at t plus 2. And finally, [38] some simplification. This is telling us the values of all interactive variables at all times. If we used real time instead of natural time, and made different assignments take different amounts of time, there would be times that the value of an interactive variable is not known.

[39] Now I want to look at a very real example, taken from the specification of a thermostat for a gas burner. This thermostat [40] goes with three other devices, all running in parallel. The thermometer tells the temperature of a room. The control tells the desired temperature. And the burner heats the room. The [41] inputs to the thermostat are: temperature, which belongs to the thermometer; desired, which belongs to the control, and flame, a binary value that comes from a flame sensor in the burner and tells whether there is a flame. These are all interactive variables because the thermostat needs their current values. The [42] outputs from the thermostat are: binary gas, to turn the gas on and off in the burner; and binary spark, to cause sparks in order to ignite the flame. These variables also have to be interactive because the burner needs their current values. Now let me read the [43] specification, which is written informally. Heat is wanted when the actual temperature falls epsilon below the desired temperature, and not wanted when the actual temperature rises epsilon above the desired temperature, where epsilon is small enough to be unnoticeable, but large enough to prevent rapid oscillation. To obtain heat, the spark should be applied to the gas for at least 1 second to give it a chance to ignite and to allow the flame to become stable. But a safety regulation states that the gas must not remain on and unlit for more than 3 seconds. Another regulation says that when the gas is shut off, it must not be turned on again for at least 20 seconds to allow any accumulated gas to clear. And finally, the gas burner must respond to its inputs within 1 second.

That's the specification for the thermostat. Now we have to formalize it so we can write a program. [44] Here's my formalization, which is not very far from being a program. We [45] start with the gas and spark off. And then we [46] check if the temperature has fallen more than epsilon below the desired temperature. If it has, [47] then we turn on the gas and the sparks, and we do this for between 1 and 3 seconds. The informal specification says at least 1 second to give the gas a chance to ignite, but not more than 3 for safety. Then we turn the sparks off. And we [48] check if heat is still wanted, and there's a flame. If so, [49] then we should leave the gas and spark as they are; that means leaving the gas on and

the spark off. And we should leave them that way for up to 1 second, but no more, before we check again. Back up [50] here, if the room was warm enough, then we [51] leave the gas and spark off, and we check again within 1 second. And back down [52] here, the gas is on and there's no spark. If it's warm enough, or there's no flame, either way we [53] turn the gas off. We leave it off, and we leave the spark off for at least 20 seconds for safety, but not more than 21 seconds without checking again.

[54] The only part of my specification that isn't already program is the timing. Whenever it says take at least this long, we can use a little loop that does nothing but take time until enough time has passed. That's called a busy-wait loop. Whenever it says take at most this long, if execution takes too long the only way we can speed it up is to build faster hardware. But we're talking about computer instruction times of a microsecond, and specification times of 3 seconds or 1 second or 21 seconds, so there's no problem.

I can't be sure that my formal specification is exactly what the person who wrote the informal specification had in mind. You can never be sure of that. Informal specifications can be ambiguous, or even self-contradictory. This one says in one place that we have to wait 20 seconds, and in another place that we have to respond to inputs within 1 second. The formal specification is unambiguous, and consistent, which means implementable. Whether it's right, I don't know.