CMU 15-671 | Models of Software Systems | Fall 1997

Shared Memory Model

Garlan | Handout 13 | 22 October 1997

This handout addresses the question "How do you model a shared memory model of concurrency with state machines?"

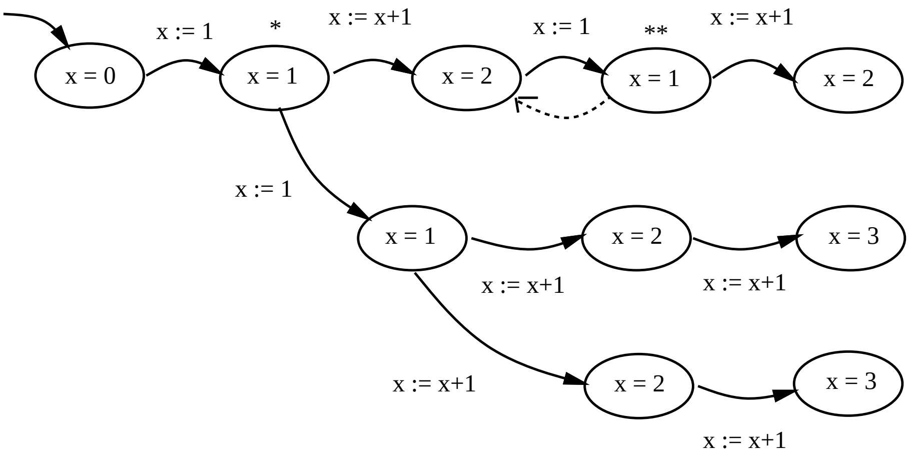
Let's consider the simple example from the Concurrency handout. We have a single shared integer variable x, initialized to 0, and two processes, P and Q, each of which executes the same code:

```
x := 1
x := x + 1
```

We have these interleavings:

$x := 1 \quad (P)$	$x := 1 \quad (P)$	$x := 1 \quad (P)$
$x := x+1 \quad (P)$	$x := 1 \quad (Q)$	$x := 1 \quad (Q)$
$x := 1 \quad (Q)$	$x := x+1 \quad (P)$	$x := x+1 \quad (Q)$
$x := x+1 \quad (Q)$	$x := x+1 \quad (Q)$	$x := x+1 \quad (P)$
$\{x = 2\}$	$\{x = 3\}$	$\{x = 3\}$

plus all three symmetric cases. Here is part of the state transition diagram:



where each statement labels a state transition arrow and the current value of x labels each node. Note two things:

First, having an arrow (indicated as a dashed line in the diagram) from ** to its predecessor rather than to a new node with the same label would be wrong because we would introduce a loop, and thus an infinite trace (one we do not expect for the behavior of this example). It's wrong because the nodes labeled $x = 1$ are really different. The $x = 1$ node with * next to it corresponds to the state we get to after P does its first action but Q has not done anything yet. The $x = 1$ node with ** next to it corresponds to the state after P has completed and Q has done its first action. Thus, we need to keep track of what P and Q have done already. Thus, the state transition diagram is missing some information— because although the two nodes are labeled the same they represent different states in the computation. Thus, we need to keep track of this missing information. In general, we need to keep track of where P and Q are in their respective code sequences.

The obvious and natural way to keep track of this information is to model explicitly the *program counter* of each concurrent process in our system. Thus, as part of each state in our state machine model we include a program counter for each process. As part of the effect of each state transition, exactly one of the program counters gets incremented.

Second, just as we now have a way to distinguish between nodes that look similar, we need a way to distinguish between state transitions that look similar. As we have already been doing informally, we simply tag each statement with the name of the process that executes it.

1 Combining State Machines

We first model each process as a (sequential) state machine. Thus, we have $P = (S_P, I_P, A_P, \delta_P)$ and $Q = (S_Q, I_Q, A_Q, \delta_Q)$. The question is "What is an appropriate state machine model for $P \,||\, Q$?" Before we answer this question, reconsider the example above, where we have (fortuitously) the simpler case that $A_P = A_Q$. You might like a property like[1]:

$$traces(P \,||\, Q) = traces(P) \cap traces(Q)$$

but that's clearly not true here! That is, there exists a trace in $traces(P \,||\, Q)$ that is not even in $traces(P)$ or $traces(Q)$.

What's going on?

1. We need to divide the state variables, $vars(P_i)$, of each process, P_i, into *locals* and *globals*. The locals are those that are affected by only that process; the globals are those that are shared with all other processes and can be changed by any of them.

2. For each process, P, we need to add its program counter, pc_P, to its set of locals.

Note that we are assuming each state is a mapping from variables to values; here, $vars(P_i)$ is the same set as Var in Handout on State Machine Basics for one machine. Since we have more than one process, we need to distinguish between the different variable sets. For simplicity, however, we will assume the same value set, Val, over which all the different types of variables range.

In what follows, to simplify the notation, for state s and variable set V, I'll write $s \upharpoonright V$ to stand for $dom(s) \upharpoonright V$ (restriction of the domain of state s to just the variables in V). I'll also write $V = V'$ to stand for $\forall\, x \in V . x = x'$ (all variables in V stay the same in value).

Now we can define $P \,||\, Q = (S_{P||Q}, I_{P||Q}, A_{P||Q}, \delta_{P||Q})$ as follows:

- The states:

$$\begin{aligned} S_{P||Q} &= \{(locals(P) \times locals(Q) \times globals(P, Q)) \twoheadrightarrow\!\!\!\!\!\!\!\!+ \;\; Val\} \\ &\quad \text{where} \\ &\qquad globals(P, Q) = vars(P) \cap vars(Q) \\ &\qquad locals(P) = vars(P) \setminus globals(P, Q) \\ &\qquad locals(Q) = vars(Q) \setminus globals(P, Q), \text{ and} \\ &\qquad pc_P \in locals(P) \wedge pc_Q \in locals(Q) \end{aligned}$$

  The sets of locals of each process are mutually disjoint and each set of locals is disjoint with the globals. The intuition is that P can freely change its own locals but not anyone else's and anyone can change any global.

[1] CSP enjoys such a property.

- The initial states:

$$I_{P||Q} = \{i \in S_{P||Q} \mid i \restriction vars(P) \in I_P \wedge i \restriction vars(Q) \in I_Q\}$$

  The initial states of the composite machine is the set of all states that are initial when projected onto the domain of each of the constituent state machines.

- The actions:

$$A_{P||Q} = A_P \cup A_Q$$

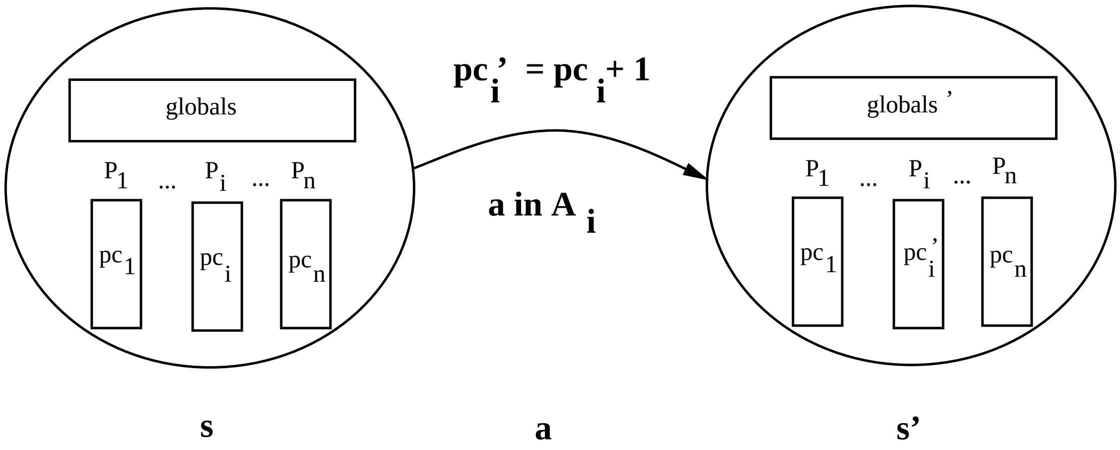
  The set of actions of the composite machine is simply the union of the sets of actions of the constituent state machines. Note that if we tagging each action with a process name, we have $A_P \cap A_Q = \emptyset$. (If we want shared actions, then we would not need to pull this trick of tagging each action.)

- The state transition relation:

$$\begin{array}{rl} \delta_{P||Q} = \{(s, a, s') \in S_{P||Q} \times A_{P||Q} \times S_{P||Q} \mid & \\ & (a \in A_P \wedge (s \restriction vars(P), a, s' \restriction vars(P)) \in \delta_P \wedge pc'_P = pc_P + 1) \wedge locals(Q) = locals'(Q) \\ \vee & \\ & (a \in A_Q \wedge (s \restriction vars(Q), a, s' \restriction vars(Q)) \in \delta_Q \wedge pc'_Q = pc_Q + 1 \wedge locals(P) = locals'(P))\} \end{array}$$

  A step, (s, a, s'), of the composite machine is a step of one of the two machines: if for P, then its *pc* gets bumped, and all locals of Q stay the same; similarly, if for Q.

  The generalization from two to i processes is straightforward. A picture of a single step transition of a composite machine with i processes looks like the following, where process P_i gets to change the globals as well as its own locals. The locals of all other processes remain the same.



2 Commentary

This model of a combined state machine works but it's not so nice. Why? It's *not compositional.* We don't have the property that

$$traces(P \mid\mid Q) = traces(P) \cap traces(Q)$$

or even that

$$traces(P \mid\mid Q) = traces(P) \cup traces(Q)$$

Moreover, we can't say

$$Inv(P) \wedge Inv(Q) \Rightarrow Inv(P \mid\mid Q)$$

That is, a property, *Inv*, that holds for both P and Q does not necessarily hold for its composition.

A property that does hold of $P \mid\mid Q$ is a *global* invariant. It's called "global" because it's a predicate that can involve, in general, the program counters of the individual processes as well as the global variables.

Nevertheless, it is a common model for a programming languages that support a shared memory model of concurrency. For example, any language that has a Threads interface such as C-Threads, Modula-2++, Modula-3, or any operating system that supports lightweight threads of control (e.g., Mach) is based on this model.

People also design systems with this model in mind. For example, Linda's tuple space is based on a so-called "blackboard" model, which is a shared resource; most transactional systems assume either physically centralized database or a globally-viewed distributed database; file systems like Andrew give the illusion of shared files and directories; user interfaces like X windows need to manage shared real estate. So, there are real-world systems that are very naturally modeled as shared memory-systems.