Computer Science and State Machines

Leslie Lamport

8 June 2008

Contribution to a Festschrift honoring
Willem-Paul de Roever on his retirement.

Computation

Computer science is largely about computation. Many kinds of computing devices have been described, some abstract and some very concrete. Among them are:

- Automata, including Turing machines, Moore machines, Mealy machines, pushdown automata, and cellular automata.
- Computer programs written in a programming language.
- Algorithms written in natural language and pseudocode.
- von Neumann computers.
- BNF grammars.
- Process algebras such as CCS.

Computer scientists collectively suffer from what I call the Whorfian syndrome[1]—the confusion of language with reality. Since these devices are described in different languages, they must all be different. In fact, they are all naturally described as state machines.

State Machines

There are two ways to define *state machine*, one emphasizing the states and the other the transitions from one state to the next. I will use the simpler one that emphasizes states. For brevity, I ignore termination/liveness and consider only safety. A state machine is then specified by a set $\mathcal{S}$ of states, a set $\mathcal{I}$ of initial states, and a next-state relation $\mathcal{N}$ on $\mathcal{S}$, so $\mathcal{I} \subseteq \mathcal{S}$ and $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$. It generates all computations $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$ such that:

S1. $s_1 \in \mathcal{I}$

S2. $\langle s_i, s_{i+1} \rangle \in \mathcal{N}$, for all i.

For example, a BNF grammar can be described by a state machine whose states are sequences of terminals and/or non-terminals. The set of initial states contains only the sequence consisting of the single starting nonterminal. The next-state relation is defined to contain $\langle s, t \rangle$ iff s can be

[1] See `http://en.wikipedia.org/wiki/Sapir-Whorf_hypothesis` .

transformed to t by applying a production rule to expand a single nonterminal.

Some of the computing devices listed above have an *event* (called an "input", "output", or "action") associated with a state transition. Those devices can be represented by augmenting the state to include the last event. In other words a transition $s \xrightarrow{\alpha} t$ from state s to state t with event α can be represented as a transition from (augmented) state $\langle s, \beta \rangle$ to state $\langle t, \alpha \rangle$, where β is the event that "led to" s. (Initial states have the form $\langle s, \perp \rangle$ for a special initial event $\perp$.)

Describing all the other kinds of computing devices listed above as state machines is straightforward. Complexity results only from the innate complexity of the device, programs written in a modern programming language being especially complicated. However, representing a program in even the simplest language as a state machine may be impossible for a computer scientist suffering from the Whorfian syndrome. Languages for describing computing devices often do not make explicit all components of the state. For example, simple programming languages provide no way to refer to the call stack, which is an important part of the state. For one afflicted by the Whorfian syndrome, a state component that has no name doesn't exist. It is impossible to represent a program with procedures as a state machine if all mention of the call stack is forbidden. Whorfian-syndrome induced restrictions that make it impossible to represent a program as a state machine also lead to incompleteness in methods for reasoning about programs.

Specifying a State Machine

To use state machines, we need a language for specifying them. The languages designed by computer scientists for describing computations usually specify state machines, defining the computations by S1 and S2. A partisan of such a language will insist that it is ideal for describing any state machine. I will ignore computer scientists and use instead the language employed by every other branch of science and engineering—namely, ordinary mathematics.

In science and engineering, a set of states is usually specified by a collection of variables and their ranges, which are sets of values. A state s assigns to every variable v a value $s(v)$ in its range. For example, physicists might describe the state of a particle moving in one dimension by variables x (the particle's position) and p (its momentum) whose ranges are the set of real numbers. The state s_t at a time t is described by the real numbers $s_t(x)$

and $s_t(p)$, which physicists usually write $x(t)$ and $p(t)$.

We specify the set of initial states the way sets of states are generally described—by a boolean-valued expression containing variables and ordinary mathematical constants and operators. For the particle example, $x = 0$ specifies the set of all states s such that $s(x) = 0$ and $s(p)$ is any real number.

Because most fields of science and engineering study continuous processes, there is no standard way to describe a next-state relation. The simplest way I know to do it is with an expression that can contain primed as well as unprimed variables, the unprimed variables referring to the first state and the primed variables to the second state. For example, $(x' = x+1)\wedge(p' > x')$ specifies the relation consisting of all pairs $\langle s, t \rangle$ of states such that $t(x) = s(x) + 1$ and $t(p) > t(x)$.

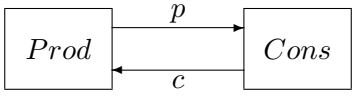
State Machines in Action

The benefits of describing state machines mathematically rather than hiding them behind computer-science languages would make a long list. It might begin with the replacement of esoteric programming logics by ordinary mathematics. For example, the Hoare triple $\{P\}S\{Q\}$ becomes the formula $P \wedge \mathcal{S} \Rightarrow Q'$, where $\mathcal{S}$ is the relation on states described by the program statement S and Q' is formula Q with each variable primed. Instead of compiling such a list, I consider one nice little example—two algorithms that appear unrelated until they are expressed mathematically as state machines.

The first algorithm is described by this simple program $\mathcal{X}$ that runs forever, alternately performing the operations $\mathcal{P}$ and $\mathcal{C}$.

$\mathcal{X}$: **loop** $\mathcal{P}$; $\mathcal{C}$ **endloop**

The second algorithm is an important hardware protocol called *two-phase handshake*, illustrated by this diagram.



The "wires" p and q can assume the values 0 and 1; the arrows indicate that p is set by process *Prod* and c is set by process *Cons*. The processes synchronize using p and c so they take turns executing operations $\mathcal{P}$ and $\mathcal{C}$. Their protocol can be described as follows, where p and c are initially equal

and $\oplus$ is defined to be addition modulo 2 (known to hardware designers as 1-bit exclusive-or).

$$\begin{array}{l} \mathcal{Y}: \ \textbf{process}\ \mathit{Prod}: \ \textbf{whenever}\ p = c\ \textbf{do}\ \mathcal{P}\ ;\ p := p \oplus 1\ \ \textbf{end} \\ \qquad\quad \| \\ \qquad \textbf{process}\ \mathit{Cons}: \ \textbf{whenever}\ p \neq c\ \textbf{do}\ \mathcal{C}\ ;\ c := c \oplus 1\ \ \textbf{end} \end{array}$$

It is easy to see, though not completely obvious, that $\mathcal{Y}$ alternately performs $\mathcal{P}$ and $\mathcal{C}$ operations, just like $\mathcal{X}$. From the state machines' pseudocode descriptions, this seems coincidental. The mathematical descriptions of these state machines reveal that it is no coincidence. Starting from $\mathcal{X}$, we can derive $\mathcal{Y}$ mathematically.

For simplicity, assume $\mathcal{P}$ and $\mathcal{C}$ to be atomic operations. They are then described by relations between primed and unprimed variables. To avoid introducing new symbols, let $\mathcal{P}$ and $\mathcal{C}$ also denote these two mathematical relations. Let $\mathit{var}_{\mathcal{PC}}$ be the set of variables that occur in these relations.

To describe program $\mathcal{X}$ as a state machine, we must introduce a variable to represent the control state—part of the state not described by program variables, so to victims of the Whorfian syndrome it doesn't exist. Let's call that variable pc, which we assume is not in $\mathit{var}_{\mathcal{PC}}$. The state variables of $\mathcal{X}$ are therefore pc and the variables in $\mathit{var}_{\mathcal{PC}}$. Since $\mathcal{P}$ and $\mathcal{C}$ are atomic operations, each executed as a single step, the variable pc assumes just two values. Let those values be 0 and 1. State machine $\mathcal{X}$ then has initial predicate $\mathit{Init}_{\mathcal{X}}$ and next-state relation $\mathit{Next}_{\mathcal{X}}$ defined as follows, where $\mathit{Init}_{\mathcal{PC}}$ specifies the initial values of the variables in $\mathit{var}_{\mathcal{PC}}$.

$$\begin{array}{lcl} \mathit{Init}_{\mathcal{X}} & \triangleq & (pc = 0) \ \wedge \ \mathit{Init}_{\mathcal{PC}} \\ \mathit{Next}_{\mathcal{X}} & \triangleq & \quad ((pc = 0) \ \wedge \ \mathcal{P} \ \wedge \ (pc' = 1)) \\ & & \vee \ ((pc = 1) \ \wedge \ \mathcal{C} \ \wedge \ (pc' = 0)) \end{array}$$

To describe $\mathcal{Y}$ as a simple state machine, we assume that the body of each process is executed as a single atomic action. Thus, when $p = c$ is true, process *Prod* both executes $\mathcal{P}$ and increments p as one step. There is then no control state, and the state variables are p, c, and the variables in $\mathit{var}_{\mathcal{PC}}$. The initial predicate and next-state relation of $\mathcal{Y}$ are

$$\begin{array}{lcl} \mathit{Init}_{\mathcal{Y}} & \triangleq & (p = c) \ \wedge \ \mathit{Init}_{\mathcal{PC}} \\ \mathit{Next}_{\mathcal{Y}} & \triangleq & \mathit{Prod} \ \vee \ \mathit{Cons} \end{array}$$

where formulas *Prod* and *Cons*, which describe the two processes, are defined by:

$$\begin{aligned} \mathit{Prod} &\triangleq (p = c) \wedge \mathcal{P} \wedge (p' = p \oplus 1) \wedge (c' = c) \\ \mathit{Cons} &\triangleq (p \neq c) \wedge \mathcal{C} \wedge (c' = c \oplus 1) \wedge (p' = p) \end{aligned}$$

The mathematical relation between these two state machines is simple:

$\mathcal{Y}$ is obtained from $\mathcal{X}$ by substituting $p \oplus c$ for pc.

Substituting an expression for a variable is a basic and powerful mathematical operation. Let us now see exactly how we derive $\mathcal{Y}$ from $\mathcal{X}$ by this substitution.

For any formula F, let $\overline{F}$ be the formula obtained from F by this substitution. For example, $\overline{pc'}$ equals $(p \oplus c)'$, which equals $p' \oplus c'$. It is easy to see that

$$\overline{pc} = \begin{cases} 0 & \text{if } p = c \\ 1 & \text{if } p \neq c \end{cases}$$

from which we obtain

$$\begin{aligned} \overline{\mathit{Init}_{\mathcal{X}}} &\triangleq (p = c) \wedge \mathit{Init}_{\mathcal{PC}} \\ \overline{\mathit{Next}_{\mathcal{X}}} &\triangleq \mathit{Pr} \vee \mathit{Co} \end{aligned}$$

where

$$\begin{aligned} \mathit{Pr} &\triangleq (p = c) \wedge \mathcal{P} \wedge (p' \neq c') \\ \mathit{Co} &\triangleq (p \neq c) \wedge \mathcal{C} \wedge (p' = c') \end{aligned}$$

The formulas $\overline{\mathit{Init}_{\mathcal{X}}}$ and $\overline{\mathit{Next}_{\mathcal{X}}}$ are the initial predicate and next-state relation of a state machine $\overline{\mathcal{X}}$ whose states are the states of $\mathcal{Y}$. We first consider its relation to state machine $\mathcal{X}$.

Define a mapping Ψ from states of $\mathcal{Y}$ to states of $\mathcal{X}$ by letting $\Psi(s)$ assign the same values to the variables in $\mathit{var}_{\mathcal{PC}}$ as s, and letting it assign to pc the value $s(p) \oplus s(c)$. (Recall that $s(p)$ and $s(c)$ are the values assigned to p and c by state s.) Extend Ψ to a mapping on computations (sequences of states) by letting $\Psi(s_1 \rightarrow s_2 \rightarrow \ldots)$ equal $\Psi(s_1) \rightarrow \Psi(s_2) \rightarrow \ldots$. It follows easily from our definition of $\overline{F}$ that a formula $\overline{F}$ is true of state s of $\mathcal{Y}$ iff F is true of state $\Psi(s)$ of $\mathcal{X}$. Similarly, a relation $\overline{R}$ is true of a pair $\langle s_1, s_2 \rangle$ of states of $\mathcal{Y}$ iff R is true of $\langle \Psi(s_1), \Psi(s_2) \rangle$. It follows that a sequence σ of states of $\mathcal{Y}$ is a computation of the state machine $\overline{\mathcal{X}}$ iff $\Psi(\sigma)$ is a computation of $\mathcal{X}$.

Let us now consider the disjuncts of the next-state relation $\overline{Next_{\mathcal{X}}}$, starting with Pr. Because p and c assume only the values 0 and 1, $p = c$ implies

$$\begin{array}{rcl} p' \neq c' & \equiv & ((p' = p \oplus 1) \ \wedge \ (c' = c)) \\ & \vee & ((p' = p) \quad \wedge \ (c' = c \oplus 1)) \end{array}$$

This implies

$$\begin{array}{rcl} Pr & \equiv & ((p = c) \wedge \mathcal{P} \wedge (p' = p \oplus 1) \ \wedge \ (c' = c)) \\ & \vee & ((p = c) \wedge \mathcal{P} \wedge (p' = p) \quad \wedge \ (c' = c \oplus 1)) \end{array}$$

A Pr step therefore either increments p and leaves c unchanged (satisfying the first disjunct) or else increments c and leaves p unchanged (satisfying the second disjunct). If we want an algorithm in which the process that executes $\mathcal{P}$ modifies only p, then we must allow only the first possibility, eliminating the second disjunct. We are left with the first disjunct, which equals $Prod$. A similar calculation shows that we obtain $Cons$ from Co by eliminating a disjunct that modifies p and leaves c unchanged. This leads us to a state machine with initial predicate $\overline{Init_{\mathcal{X}}}$ and next-state predicate $Prod \vee Cons$, which is precisely the state machine $\mathcal{Y}$.

Our derivation shows that $Prod$ implies Pr and $Cons$ implies Co. Hence, $Next_{\mathcal{Y}}$ implies $\overline{Next_{\mathcal{X}}}$. Since $Init_{\mathcal{Y}}$ equals $\overline{Init_{\mathcal{X}}}$, we deduce that any computation σ of $\mathcal{Y}$ is a computation of $\overline{\mathcal{X}}$. We have already seen that σ is a computation of $\overline{\mathcal{X}}$ iff $\Psi(\sigma)$ is a computation of $\mathcal{X}$. Hence, if σ is any computation of $\mathcal{Y}$, then $\Psi(\sigma)$ is a computation of $\mathcal{X}$. Because the states s and $\Psi(s)$ assign the same values to the variables in $var_{\mathcal{PC}}$, this means that σ has the same $\mathcal{P}$ and $\mathcal{C}$ steps as $\Psi(\sigma)$. Thus, we deduce that the derived protocol $\mathcal{Y}$ produces the same sequence of $\mathcal{P}$ and $\mathcal{C}$ operations as does $\mathcal{X}$. Since it is obvious that $\mathcal{X}$ alternately executes $\mathcal{P}$ and $\mathcal{C}$ operations, this shows that $\mathcal{Y}$ does too. In other words, this shows that $\mathcal{Y}$ is correct by construction.

When presenting this kind of derivation, it is conventional to pretend that it leads to the discovery of the resulting protocol. I presented $\mathcal{Y}$ before the derivation to make it easier to see where we were heading. This allowed me to "cheat" by letting pc assume the convenient values 0 and 1. Had I chosen two arbitrary values a and b instead, we would have substituted

IF $p = c$ THEN a ELSE b

for pc. A simple calculation would have shown

$$\overline{pc} \ = \ \begin{cases} a & \text{if } \ p = c \\ b & \text{if } \ p \neq c \end{cases}$$

From that point, the derivation would have proceeded exactly as before, with the same formulas $\overline{Init_{\mathcal{X}}}$ and $\overline{Next_{\mathcal{X}}}$.

A Lesson

Using ordinary mathematics, we have derived the simple but useful protocol $\mathcal{Y}$ from the trivial algorithm $\mathcal{X}$ by substituting $p \oplus c$ for pc. We could do this because we represented these algorithms as state machines and we described the state machines using ordinary mathematics.

The pseudocode descriptions probably seem more natural to most computer scientists. But how could our derivation possibly have been done from those descriptions? How do we substitute for a variable pc that doesn't appear in the pseudocode? Even if pc did appear as a variable, what would it mean to substitute an expression for it in an assignment statement $pc := \ldots$?

Quite a number of formalisms have been proposed for specifying and verifying protocols such as $\mathcal{Y}$. The ones that work in practice essentially describe a protocol as a state machine. Many of these formalisms are said to be mathematical, having words like *algebra* and *calculus* in their names. Because a proof that a protocol satisfies a specification is easily turned into a derivation of the protocol from the specification, it should be simple to derive $\mathcal{Y}$ from $\mathcal{X}$ in any of those formalisms. (A practical formalism will have no trouble handling such a simple example.) But in how many of them can this derivation be performed by substituting for pc in the actual specification of $\mathcal{X}$? The answer is: very, very few. Despite what those who suffer from the Whorfian syndrome may believe, calling something mathematical does not confer upon it the power and simplicity of ordinary mathematics.