Massachusetts Institute of Technology Handout 11
6.042J/18.062J: Mathematics for Computer Science March 4, 1997
Professor David R. Karger

State Machines

(by Nancy Lynch)

1 Introduction

In this handout, we present important discrete structures called *state machines*. State machines are used in computer science as the basis for modelling algorithms and physical machines, in order to give precise specifications of their behavior and in order to verify their properties. They are useful for both software and hardware.

We first present the definition of a state machine, and the definitions of important related concepts like *executions* and *reachable states* of a state machine. We then give some basic examples that illustrate how state machines can be used to represent computational processes. These examples should demonstrate the utility of decomposing the states of a state machine by means of *state variables*, for convenience in representation and manipulation. We also define and illustrate *deterministic* and *nondeterminisitic* state machines.

We then introduce the key notion of *invariants*; an invariant is a property that is always true in all the reachable states of a state machine. Invariants are the main tool used in practice, in formal reasoning about algorithms and machines. We show how invariants can be proved, using mathematical induction. We show some invariants that can be proved directly, and others that require strengthening of the inductive hypothesis in order for an inductive proof to work. We also illustrate proofs of invariants in batches rather than one at a time.

Next, we present a collection of more interesting examples, taken from several domains studied in this course. First, we consider numerical calculations, including Euclid's GCD algorithm, and a fast multiplication algorithm. Second, we examine poset-based computations, including binary search and bubble sort. Third, we present some graph algorithms, including one that maintains the property that the graph is acyclic and one that carries out a breadth-first search. Finally, we present some state machines derived from common distributed algorithms – a leader election algorithm and a well-known communication protocol.

At the end, we say a few words about some more advanced topics: safety properties vs. liveness properties, temporal logic, progress functions, and simulation relations.

2 Definitions: State Machine, Executions and Reachability

A *state machine* is defined to be a triple (Q, Q_0, δ), where:

1. Q is a nonempty (finite or infinite) set of *states*.
2. $Q_0 \subseteq Q$ is a nonempty subset of the states known as the *start states*.
3. $\delta \subseteq Q \times Q$ is a binary relation over Q known as the *transition relation*.

Note: You shouldn't confuse state machines with the *finite state machines* studied in 6.045 and 6.840 and elsewhere. These machines need not have only finitely many states. Formally, a state machine is similar to a digraph – the differences are that the set Q can be infinite, and that there are distinguished start states.

The whole point of a state machine is that it *executes*, producing *executions*. That is, it starts from any start state and moves along, following transitions. In this way, it models a computational process.

Formally, an *execution fragment* of a state machine is any finite or infinite sequence $q_0, q_1, \ldots$ such that $(q_i, q_{i+1}) \in \delta$ for every $i \geq 0$.
An *execution* is an execution fragment in which the initial state q_0 is an element of Q_0. A state in Q is said to be a *reachable* state if it is the last state of some finite execution.

We give some simple examples. We begin with a very simple infinite-state machine having one initial state, and having no choices of what to do next.

Example 1 Counter

We define:

- $Q = \mathbb{N}$.
- $Q_0 = \{0\}$.
- $\delta = \{(n, n+1) | n \in \mathbb{N}\}$.

Intuitively, this models a counting process that starts with value 0 and increments (forever). The reachable states are all of $Q = \mathbb{N}$. Each natural number n is reached by a unique execution with n transitions ($n + 1$ states). There is only one execution containing any particular number of transitions.

The counter in Example 1 is an example of a special kind of state machine that we call *deterministic*. This means that it satisfies the following two properties:

1. It has only a single start state: $|Q_0| = 1$.
2. Each state has at most one successor: for each $q \in Q$, there is at most one q' such that $(q, q') \in \delta$.

If a state machine is not deterministic, then we say it is *nondeterministic*. This means it has choices, either of starting state, of transitions, or of both. The following is an example of a nondeterministic state machine:

Example 2 Linear hopscotch, with steps of 12 and 18

This example represents a "hopscotch" game played on a two-way-infinite one-dimensional court. The player starts on the square numbered 0, and is allowed to hop either left or right, by either 12 or 18 squares.

Formally, we define:

- $Q = \mathbb{Z}$.
- $Q_0 = \{0\}$.
- $\delta = \{(i, i+12) | i \in \mathbb{Z}\} \cup \{(i, i-12) | i \in \mathbb{Z}\} \cup \{(i, i+18) | i \in \mathbb{Z}\} \cup \{(i, i-18) | i \in \mathbb{Z}\}$.

This state machine has many different executions, all of which lead to states that are multiples of $gcd(12, 18)$, which is 6.

We can also modify this machine slightly to allow multiple start states. For example, we could allow Q_0 to be the infinite set $\{i | i = 0 \pmod 6\}$, or the two-element set $\{0, 2\}$. Note that in the former case, all the reachable states are still multiples of 6, but that is not true in the latter case.

In most examples of state machines that occur in practice, the state has *structure*: it can be described as containing different pieces of information, which can be represented as values of *state variables*. (In this way, state machines differ from digraphs, whose vertices are usually thought of as unstructured.) The state variable structure leads to concise, understandable representations.

Example 3 2-dimensional hopscotch, with steps of 12 and 18 east or west, and steps of 9 and 12 north or south

Here, Q consists of values for two *variables*:

> *east-west* $\in \mathbb{Z}$, initially 0
> *north-south* $\in \mathbb{Z}$, initially 0

Formally, Q is the Cartesian product, $\mathbb{Z} \times \mathbb{Z}$. Also, Q_0 consists of the single pair, $(0, 0)$.

We define the transition relation δ by a collection of code fragments. Namely, all the transitions in δ are of the following forms:

move-east(12)
  Effect:
    $east\text{-}west := east\text{-}west + 12$

move-east(18)
  Effect:
    $east\text{-}west := east\text{-}west + 18$

move-west(12)
  Effect:
    $east\text{-}west := east\text{-}west - 12$

move-west(18)
  Effect:
    $east\text{-}west := east\text{-}west - 18$

move-north(9)
  Effect:
    $north\text{-}south := north\text{-}south + 9$

move-north(12)
  Effect:
    $north\text{-}south := north\text{-}south + 12$

move-south(9)
  Effect:
    $north\text{-}south := north\text{-}south - 9$

move-south(12)
  Effect:
    $north\text{-}south := north\text{-}south - 12$

The names we associate with the transitions are called *labels*. They have no formal significance here, but are used because they are convenient for classifying the transitions. Formally, δ consists of all pairs of states that are described by any of the code fragments. A code fragment describes a set of pairs of states, in an obvious way. For example, the first code fragment, for *move-east*(12), denotes all pairs of states (q, q'), where *east-west* in q' is 12 more than it is in q, and *north-south* is the same in q' as it is in q. Written concisely: $q'.east\text{-}west = q.east\text{-}west + 12$, and $q'.north\text{-}south = q.north\text{-}south$.

The transitions above are described using assignment statements. This should be easy notation for programmers to understand. Alternatively, the transitions in each code fragment can be described using equations like the two just above.

There is one more important wrinkle: When using code fragments to represent state transitions, sometimes it's convenient to give a predicate restricting when the code fragment is allowed to be applied:

Example 4 2-dimensional hopscotch on the northeast quadrant only

This is very similar to the machine in Example 3, only now the two state components *east-west* and *north-south* take on nonnegative integer values only, that is, values in $\mathbb{N}$.

Now Q is the Cartesian product $\mathbb{N} \times \mathbb{N}$, and Q_0 consists of the single pair $(0, 0)$. Now we have to be careful in the definition of δ – some of the named sets of transitions, in particular, the *move-west* and *move-south* transitions, cannot occur in all situations. For example, the *move-west*(12) transitions can only occur when $east\text{-}west \geq 12$. The revised code fragments can be written as follows:

move-west(12)
  Precondition:
    $east\text{-}west \geq 12$
  Effect:
    $east\text{-}west := east\text{-}west - 12$

move-west(18)
  Precondition:
    $east\text{-}west \geq 18$
  Effect:
    $east\text{-}west := east\text{-}west - 18$

move-south(9)
  Precondition:
    $north\text{-}south \geq 9$
  Effect:
    $north\text{-}south := north\text{-}south - 9$

move-south(12)
  Precondition:
    $north\text{-}south \geq 12$
  Effect:
    $north\text{-}south := north\text{-}south - 12$

The precondition in each of these code fragments is a predicate on the state variables restricting when the transitions denoted by this code fragment are permitted to occur. Formally, the transitions denoted by the fragment are all (q, q') for which the precondition predicate is true of q and for which q' is obtained from q by executing the effect code.

State machines can be used to describe most computing processes, including both algorithms and computing machines. The state variable structure is usually crucial for obtaining understandable respresentations.

3 Invariants

Now we are ready to introduce the main practical method for reasoning about the correctness of algorithms and physical machines represented as state machines – the method of *invariants*. This method involves arguing that the machine always preserves properties of the state known as *invariants*. This is often the key to understanding what the machine does, and why.

Definition 3.1 *An* invariant *of a state machine M is any predicate on the states of M that is true of all the reachable states of M.*

Example 5 In Example 3, the general 2-dimensional hopscotch example with initial state $(0, 0)$, the statement that *east-west* is a multiple of 6, and the statement that *north-south* is a multiple of 3, are both invariants.

These statements give a lot of insight into what this machine does.

How can we prove that a state machine satisfies an invariant? We have to show that the invariant is true of all reachable states. Reachable states are those that are reached by means of some number of transitions, beginning from a start state. So, it is possible to prove the invariant *by induction on the number n of transitions* leading to a reachable state. The base case, $n = 0$, involves 0 transitions, i.e., it involves showing that the property is true in all start states. The inductive step, involves showing that the property is preserved by all transitions.

The following example shows the proof of an invariant.

Example 6 2-dimensional hopscotch

We prove the invariants stated in Example 5, using induction.

Base: 0 transitions. In the unique initial state, both components are 0. Clearly, this means that the *east-west* component is a multiple of 6 and the *north-south* component is a multiple of 3.

Inductive step: Suppose that the property is true for all states reachable after n transitions, and now consider an arbitrary state reachable after $n+1$ transitions. That state can be reached by an execution that consists of n transitions, plus one last transition. Call the state after n steps q and the state at the end of the execution q'. By inductive hypothesis, we may assume that the invariants are true for q. That is, $q.\mathit{east\text{-}west}$ is a multiple of 6 and $q.\mathit{north\text{-}south}$ is a multiple of 3. We must show that $q'.\mathit{east\text{-}west}$ is a multiple of 6 and $q'.\mathit{north\text{-}south}$ is a multiple of 3.

We consider cases based on the type of transition. (This is the usual way to organize the inductive step, when the transitions are represented by code fragments.) Here, all the cases are basically the same, so we just do the first case, $\mathit{move\text{-}east}(12)$.

In this case, since $q.\mathit{east\text{-}west}$ is a multiple of 6 and $q'.\mathit{east\text{-}west} = q.\mathit{east\text{-}west}+12$, we also have that $q'.\mathit{east\text{-}west}$ a multiple of 6. Also (don't forget), since $q.\mathit{north\text{-}south}$ is a multiple of 3 and $q'.\mathit{north\text{-}south} = q.\mathit{north\text{-}south}$, we also have $q'.\mathit{north\text{-}south}$ a multiple of 3. This completes the inductive step and shows that the property is true in all reachable states, as needed.

In the rest of these notes, we do more examples, to illustrate the range of applicability of the method. We show how state machines can be used to represent typical computational processes, and how interesting properties of those processes can be proved as invariants, by induction. These examples involve number theory, posets, graphs, and distributed systems.

4 Number-Theoretic State Machines

We give two examples here: Euclid's GCD algorithm, and a fast multiplication algorithm.

4.1 Euclid's GCD Algorithm

Recall from an earlier lecture what this algorithm does: It takes inputs $x, y \in \mathbb{N}$, initially arbitrary, subject to the restrictions that they are not both 0, and that $x \geq y$. It returns $gcd(x, y)$.

We can model this algorithm by a state machine as follows. The state set Q consists of values for state components:

$x \in \mathbb{N}$
$y \in \mathbb{N}$
$x_1 \in \mathbb{N}$

$y_1 \in \mathbb{N}$
$answer \in \mathbb{N} \cup \{null\}$

Here, x and y are the inputs, for which we want to find the GCD. They are initially arbitrary, subject to the conditions that they are not both 0 and that $x \geq y$. The next two components, x_1 and y_1, are auxiliary components, which keep track of the current values on which we are working. Initially, $x_1 = x$, $y_1 = y$ The final component, *answer* is intended to hold the final answer. It can hold either an answer or *null*, which represents that the answer has not yet been found. Initially, it has value *null*.

The start states Q_0 are defined by the allowed initial values, as described above.

The transition relation δ is defined by the following two code fragments. In this algorithm, only two kinds of things can happen: recursive calls, and final decisions.

recursive-call
Precondition:
$answer = null$
$y_1 \neq 0$
Effect:
$(x_1, y_1) := (y_1, x_1 \bmod y_1)$

decision
Precondition:
$answer = null$
$y_1 = 0$
Effect:
$answer := x_1$

In a typical execution, a sequence of *recursive-call* transitions occur, followed by a single *decision* transition.

We would like to prove the following property of all reachable states of this machine:

"If $answer \neq null$ then $answer = gcd(x, y)$."

This says that the state machine, if it ever reaches an answer, reaches the correct answer. This is the statement of *partial correctness* for the Euclidean GCD algorithm. (Here we ignore the other aspect of correctness – termination.)

It turns out that, although the preceding property is in fact true in all reachable states, we cannot prove it as it stands, using induction. This is because the property is not strong enough to be preserved by all the transitions. In particular, we cannot prove that this property by itself is preserved by a *decision* transition, which converts *answer* from *null* to non-*null*; to do this, we would need to know something about the reachable states in which $answer = null$. As often happens in inductive proofs, we need to strengthen the inductive hypothesis, in this case to say something about the case where $answer = null$.

Here, what works is a two-part invariant, where the first part is the property above and the second part is an auxiliary invariant. The property we try to prove by induction is the *conjunction* of these two properties.

1. If $answer \neq null$ then $answer = gcd(x, y)$.
2. $gcd(x_1, y_1) = gcd(x, y)$.

The auxiliary invariant says that the GCD of the current values is the same as that of the original inputs.

Now we can prove this two-part invariant by induction on the number of steps in an execution.

Proof. (Of two-part invariant)

Base: 0 transitions. In any start state, $answer = null$, so the first property is true vacuously. For the second property, since $x_1 = x$ and $y_1 = y$, obviously $gcd(x_1, y_1) = gcd(x, y)$, as needed.

Inductive step: Assume that both properties are true for all states reachable after n transitions, and consider an arbitrary state q' reachable after $n+1$ transitions. As before, fix an $n+1$-step execution leading to q' and let q denote the state just before the last transition. By inductive hypothesis, we can assume that both properties are true for q; we must prove them for q'. We consider cases based on the type of the last transition: *recursive-call* or *decision*.

recursive-call:

Then the precondition in the code fragment says that, just before the last step, $answer = null$; this step keeps it $null$. So the first property is again true vacuously. For the second property, we must show that $gcd(x_1, y_1) = gcd(x, y)$ after the step. We may assume this before the step, by inductive hypothesis. Since we are talking about the values of variables in different states, it is helpful to have notation to indicate which state we are talking about. We use primes for the values for the values in state q', and leave off the primes for state q, as follows: we assume that $gcd(x_1, y_1) = gcd(x, y)$, and must prove that $gcd(x_1', y_1') = gcd(x', y')$.

This conclusion follows using algebraic manipulation, and the following facts:

1. $gcd(x_1, y_1) = gcd(x, y)$. We assumed this just above.

2. $x = x'$ and $y = y'$.
   The state variables x and y remain unchanged by the transition.

3. $gcd(x_1, y_1) = gcd(y_1, x_1 \bmod y_1)$.
   This is an interesting fact, which we proved earlier in the course when we did the GCD algorithm for the first time. It relies on the fact that $y_1 \neq 0$, which is true by the precondition.

4. $x_1' = y_1$ and $y_1' = x_1 \bmod y_1$.
   This follows from the effect of the code fragment.

We leave it to the reader to carry out the algebra.

decision:

In this case, the precondition says that $answer = null$ and $y_1 = 0$ before the last step; after the last step, $answer \neq null$. So now we must do some real work to show the first property: we must show that $answer = gcd(x, y)$ after the step. By inductive hypothesis, we know that, before the step, $gcd(x_1, y_1) = gcd(x, y)$. Using prime notation, we must prove that $answer' = gcd(x', y')$.

This follows using algebra, from the facts:

1. $gcd(x_1, y_1) = gcd(x, y)$. This is as noted above.

2. $x = x'$, $y = y'$.

3. $gcd(x_1, 0) = x_1$.

4. $answer' = x_1$.
   This follows from the effect of the code fragment.

5. $y_1 = 0$.
   As noted above.

Again, we leave the algebra to the reader.

The second property is easy to show, because none of the variables that are mentioned in it are modified by the transition. ■

This example has illustrated the proof strategy of strengthening an invariant in order to prove it by induction. The strategy of strengthening an inductive hypothesis is very common in inductive proofs in general. It is almost always used in proofs of invariants for state machines.

4.2 Fast Multiplication

This algorithm is sometimed called "Russian peasant's multiplication".

The state set Q consists of values for the following variables; all are natural numbers.

x, y, the values to be multiplied
a, the accumulator for the answer
r, s, temporaries

Initially, x and y are arbitrary positive integers, $r = x$, $s = y$, and $a = 0$.

The transitions are of two types, based on whether the currect value of s is even or odd:

even
  Precondition:
    s is even
    $s > 0$
  Effect:
    $r := 2r$
    $s := \frac{s}{2}$

odd
  Precondition:
    s is odd
  Effect:
    $a := a + r$
    $r := 2r$
    $s := \frac{s-1}{2}$

The answer is the value in the accumulator a when $s = 0$. The correctness of this answer may not be completely obvious. To illustrate the behavior of the algorithm, consider the following example execution:

`$x$`	`$y$`	`$r$`	`$s$`	`$a$`
5	9	5	9	0
5	9	10	4	5
5	9	20	2	5
5	9	40	1	5
5	9	80	0	45

The initial values of the five variables are given in the first row. Successive rows give the values after successive transitions.

In order to see that this algorithm is correct, we would like to prove that the following statement is an invariant:

Property 1: If $s = 0$ then $a = xy$.

Of course, we need to strengthen Property 1 in order to prove it by induction. It turns out that the key additional property is:

Property 2: $xy = rs + a$

Notice how this property expresses the main idea that makes the algorithm work. It is typical for invariants to express the key intuitions behind algorithms.

It is possible to prove Properties 1 and 2 together, by induction, as in the previous examples. Alternatively, and slightly better, we can just prove Property 2 by itself, by induction, and then infer Property 1 as a corollary.

Proof. (That Property 2 is an invariant)

We use induction.

Base: The property is true in all start states because $r = x$, $s = y$ and $a = 0$.

Inductive step: Consider cases based on the type of last transition, *even* or *odd*.

even:

Doubling r while halving s doesn't change the product rs,
and a doesn't change. So it's easy to see that the equation is preserved by the transition.

odd:

This case is not quite so easy to see, so we work it out carefully using prime notation, as before. By inductive hypothesis, we may assume that $xy = rs + a$ (in the state before the transition) We must show $x'y' = r's' + a'$ (in the state after the transition).

We know that $x = x'$ and $y = y'$, so the left-hand side is unchanged by the transition. Also, $r' = 2r$, $s' = \frac{s-1}{2}$, and $a' = a + r$, by the effect of the step. So $r's' + a' = 2r\frac{s-1}{2} + a + r$. Reorganizing terms yields $rs + a$, as needed. ∎

So we have proved that Property 2 is an invariant. Now we can prove that Property 1 is an invariant, without even using an induction:

Proof. (That Property 1 is an invariant)

We already know that in all reachable states, $xy = rs + a$. We need to show that in all reachable states, if $s = 0$ then $a = xy$. We get this by just plugging in $s = 0$ in the first equation. ∎

This example has illustrated the proof strategy of strengthening a required invariant, then proving only a part of it by induction and getting the rest as a consequence (corollary).

A generalized version of this strategy is often used in more complicated cases: Sometimes to prove that a property is an invariant we must extend it to a long list of properties. Then we could prove the entire list by induction. However, it is usually possible (and nicer) to first prove a small number of the properties by themselves, using induction. After that, we can assume that these properties hold in all reachable states, while we prove another batch using another induction. We can continue in this way, proving all the properties in batches, using separate inductive arguments. Anywhere along the way, we might be able to obtain some of the other properties as direct consequences (as in the example in this section), without using induction.

5 Poset-Based State Machines

Algorithms can also manipulate other data structures besides just numbers, and state machine can be used to represent such algorithms. In this section, we consider algorithms that manipulate posets. Many such algorithms are based on comparisons between poset elements, and accomplish such tasks as searching and sorting.

5.1 Binary Search

This is a comparison-based searching algorithm. For simplicity, we consider a fixed positive integer k, and assume that the set in which we are searching contains $2^k - 1$ natural numbers. The object is to discover if a particular natural number is in the set. We assume that the elements of the set appear in an array, in order. The algorithm proceeds by comparing the input value to the middle element of the array, then to the middle of one of the halves, then to the middle of one of the quarters, ..., until the value is found or until there is nowhere left to look.

The state machine representing this algorithm has a state consisting of:

a, an array indexed by $\{1, \ldots, 2^k - 1\}$ of $\mathbb{N}$, initially arbitrary,
subject to $a(i) \leq a(j)$ if $i < j$.
$v \in \mathbb{N}$, initially arbitrary
$i \in \mathbb{N}$, initially 2^{k-1}
$jump \in \mathbb{Z}$, initially $k - 1$
$ans \in \{1, \ldots, 2^k - 1\} \cup \{unknown, no\}$, initially $unknown$

Here, v represents the input, i is an index into the array indicating the next element to be involved in a comparison, $jump$ keeps track of the (log of the) latest distance between two elements to which v has been compared, and ans contains the final answer – either an index where the element appears, or an indication that it does not appear.

The transitions can be classified according to whether the current comparison comes out as $=$, $<$, or $>$.

equals
  Precondition:
    $ans = unknown$
    $a(i) = v$
  Effect:
    $ans := i$

less
  Precondition:
    $ans = unknown$
    $v < a(i)$
  Effect:
    if $jump > 0$ then
      $jump := jump - 1$
      $i := i - 2^{jump}$
    else $ans := no$

greater
  Precondition:
    $ans = unknown$
    $v > a(i)$
  Effect:
    if $jump > 0$ then
      $jump := jump - 1$
      $i := i + 2^{jump}$
    else $ans := no$

(Partial) correctness for this algorithm can be expressed in terms of the following properties:

Property 1: If $ans = no$ then $(\forall i) v \neq a(i)$.

Property 2: If $ans = i \in \mathbb{N}$ then $a(i) = v$.

In order to prove that these are invariants, we have to strengthen them to include a property of the states in which $ans = unknown$. We define:

Property 3: If $ans = unknown$ then for all j, if $a(j) = v$, then $i - (2^{jump} - 1) \leq j \leq i + (2^{jump} - 1)$.

We prove all three properties together:

Proof. (That Properties 1, 2 and 3 are invariants)

It is useful to first prove the simple fact that $jump \geq 0$ is an invariant. This can be shown by an easy induction.

By induction. Base: In an initial state, $ans = unknown$, so the only thing to show is Property 3. But since $i = 2^{k-1}$ and $jump = k - 1$, Property 3 only claims in this case that $1 \leq j \leq 2^k - 1$, which is true because $2^k - 1$ is the size of the entire array a.

Inductive step: By cases, based on type of transition.

equals: Then after the step, $ans = i$. Since, by the precondition, $a(i) = v$, this means that Property 2 holds after the step. Properties 1 and 3 are vacuously true.

less: The precondition says that before the last transition, $ans = unknown$. Therefore, by inductive hypothesis (Property 3), we know that if $a(j) = v$, then $i - (2^{jump} - 1) \leq j \leq i + (2^{jump} - 1)$ before the last step. Since $v < a(i)$ and the array is sorted, we know that if $a(j) = v$ then actually $i - (2^{jump} - 1) \leq j < i$.

If $jump > 0$, then $ans' = unknown$. In this case, $jump' = jump - 1$ and $i' = i - 2^{jump'}$. Then $i' - (2^{jump'} - 1) = i - (2^{jump} - 1)$ and $i' + (2^{jump'} - 1) = i - 1$. It follows that $i' - (2^{jump'} - 1) \leq j \leq i' + (2^{jump'} - 1)$, which shows Property 1. Properties 2 and 3 are vacuously true.

On the other hand, if $jump = 0$ (the only other possibility), then $ans' = no$. In this case, the set of j such that $i - (2^{jump} - 1) \leq j < i$ is empty, so v does not appear in the array. This shows Property 1. Properties 2 and 3 are vacuously true.

greater: The argument is symmetric to the one for *less*.

This completes the inductive proof of the three properties. ∎

As you can see, inductive proofs of invariants can be slightly tricky. Even when the basic ideas are reasonably simple, there can be a lot of work getting the details straight. In this way, the task of proving invariants for programs is similar to the task of writing programs. There are systems that provide computer assistance in keeping track of the details in invariant assertion proofs.

5.2 Bubble Sort

This is a comparison-based sorting algorithm. It works in place. Fix $n \geq 2$, the length of the sequence to sort. The algorithm proceeds in passes $2, \ldots, n$, where pass i is devoted to placing the element that originates in position i in the proper position relative to the elements that originate in positions $1, \ldots, i-1$. This is accomplished by "bubbling" the element up using comparisons, until it reaches its proper position.

The state consists of:

a, a length n sequence of integers, with components $a(1), a(2), \ldots a(n)$, initially arbitrary.
i, j, pointers in the range $[2, n+1]$, initially 2

i keeps track of which pass we're up to. j holds the current position of the element for which we are currently seeing a proper position.

We can think of there being only one type of transition, though it has several cases.

compare
  Precondition:
    $i \leq n$
  Effect:
    if $a(j-1) \leq a(j)$ then $i, j := i+1$
    else
      interchange $a(j-1)$ and $a(j)$
      if $j > 2$ then $j := j-1$ else $i, j := i+1$

When a *compare* occurs, the element being bubbled up appears in position $a(j)$, and it is compared to $a(j-1)$. If $a(j-1) \leq a(j)$, then it means that the new element has found its proper position, and the current pass is complete; a new pass is begun by setting i and j to the next pass number $i+1$. Otherwise, the two elements are interchanged. Then the pass continues by setting j to point to the new position of the new element, unless $j = 2$ – in that case, the pass is over because the new element has reached the top, so we again go on to the next pass.

The property of interest is:

Property 1: If $i = n+1$ then a is sorted (in nondecreasing order).

In order to prove this by induction, we need additional properties to describe partial progress: First, there are two very simple properties, which can be proved easily by induction. They involve the ranges of index values.

1. $2 \leq j \leq i \leq n+1$.

2. $i = n+1$ if and only if $j = n+1$.

The next two properties express the main ideas of the algorithm.

Property 2: $a(k) \leq a(l)$ for all $k \neq j$, $l \neq j$, $1 \leq k \leq l \leq \min(i, n)$.

This says that all elements up to and including i, except possibly the jth element, are in sorted order.

Property 3: $a(j) \leq a(k)$ for all k, $j+1 \leq k \leq \min(i, n)$.

This says that the jth element is in the right place with respect to the elements in higher-numbered positions, up to and including the ith position.

Proof. (That Properties 2 and 3 are invariants)

We prove them together, by induction.

Base: Both properties are vacuously true in any start state.

Inductive step: The argument proceeds in cases, based on which clause in the effect code is followed.

Case 1: The first test comes out positive, saying $a(j-1) \leq a(j)$.

Informally, this means that the jth element is already in the right position relative to all others, so we have a total order for $a(1), \ldots, a(i)$. This is what is needed to start the next phase, phase $i+1$. Formally, we need to show that Properties 3 and 4 are true after the last step. These are both implied by the claim that $a'(1) \leq a'(2) \ldots a'(i)$ is sorted – that is, that after the last step, the array up to the position given by the *old* value of i is sorted. (This implies Property 2 because that property explicitly excludes the position given by $j' = i+1$ from consideration, and it implies Property 3 because that property is vacuous.)

So we must show that $a'(1) \leq a'(2) \ldots a'(i)$ is sorted. By inductive hypothesis, we know that $a(1) \leq a(2) \leq \ldots \leq a(j-1) \leq a(j+1) \leq \ldots \leq a(i)$ (by Property 2) and that $a(j) \leq a(k)$ for all k, $j+1 \leq k \leq i$. By the precondition, we know that $a(j-1) \leq a(j)$. These facts together imply that $a(1) \leq a(2) \leq \ldots \leq a(j-1) \leq a(j) \leq a(j+1) \leq \ldots \leq a(i)$. That is, $a(1) \leq a(2) \leq \ldots \leq a(i)$. But the elements in the array are unchanged by the step. So $a'(1) \leq a'(2) \leq \ldots \leq a'(i)$, as needed.

Case 2: The test comes out negative.

In this case, a single interchange occurs, of the elements in positions $j-1$ and j. Informally, the new element is correctly ordered with respect to all later elements, after the exchange, which yields Property 3 after the step. And all other elements up to i are still correctly sorted, which gives Property 2 after the step. Formally, there are two subcases, based on whether we are done with pass i.

We leave to the reader the task of finishing these two subcases, carefully. The argument is in the same style as the other arguments above. The argument is a bit technical – the

main difficulty is keeping the notation straight for the values of variables before and after the step. For example, we need to use facts like:
If $a(j) < a(j-1)$ then $a'(j') < a'(j'+1)$. (The primes are used here for both a and j because both the array contents and the index are changing.) ■

After Properties 2 and 3 are proved, we still need to show Property 1.

Proof. (Of Property 1)

This follows from Property 2, applied with $j = n+1$ (using one of the basic properties to deduce that also $i = n+1$). ■

6 Graph Algorithms

Algorithms can also manipulate graphs.

6.1 Acyclic Graph Maintenance

This is inspired by a distributed network resource allocation algorithm of Chandy and Misra.

The state is a directed graph G with no self-loops, represented by components V and E. Initially, it is any acyclic digraph (one with no directed cycles). There is only one type of transition: Pick a node with only incoming edges, and reverse all the edges simultaneously:

$reverse(v)$
  Precondition:
    E contains no edge of the form (v, w)
  Effect:
    let $W = \{w \in V | (w, v) \in E\}$
    $E := E \cup \{(v, w) \mid w \in W\} - \{(w, v) \mid w \in W\}$

We are interested in showing that the following is an invariant:

Property 1: Graph (V, E) is acyclic.

Proof. By induction. Base: By definition.

Inductive step: Suppose graph is acyclic before the last step. We claim that one reversal can't introduce a cycle. For if it did, then one of the new edges inserted by the reversal would have to be involved in the cycle. (This uses the inductive hypothesis, which says there are no cycles before the last step.) But each such edge starts at v, so the cycle would have to also have an edge that ends with v. But this can't be, since we are reversing *all* edges incoming to v. ■

6.2 Minimum Weight Paths ("Shortest Paths")

We give an algorithm for finding minimum-weight paths (also known as shortest paths) in a digraph. Such algorithms are used for determining best routes for messages in networks.

Once again, the state contains a digraph G, represented by components V and E. This time, it also contains v_0, a distinguished node in V. Also, each edge (v, w) in E has a nonnegative real-valued *weight*, $weight(v, w)$. (We do not assume that weights are symmetric; this $weight(v, w)$ could be different from $weight(w, v)$.) These weights are represented in a matrix.

The goal is for each node to determine the total weight of a minimum-weight path from v_0 to itself in the digraph. A minimum-weight path is also known as a "shortest path" in the literature, using a strange mix of weight and length terminology.

There is one other state component: Each node v has a path-weight estimate, $pweight(v)$, which is either a nonnegative real or ∞. Initially, $pweight(v_0) = 0$, and all the other *pweights* are ∞.

There is only one type of transition, in which one of the vertices, v, improves (lowers) its path weight estimate based on the latest estimate of one of its neighbors, w:

$improve(w, v)$
  Precondition:
    $(w, v) \in E$
    $pweight(w) + weight(w, v) < pweight(v)$
  Effect:
    $pweight(v) := pweight(w) + weight(w, v)$

This state machine is very nondeterministic – it has lots of different executions, depending on which vertices improve their estimates, based on which neighbors, in which order. What happens in the course of an execution? First, at any time, for each vertex that has a finite estimate, the estimate is in fact the total weight of some path from v_0 to itself. Second, if all vertices keep trying to improve based on all incoming neighbors, then eventually each will have as its estimate the weight of an actual minimum weight path from v_0 to itself.

The first property can be proved as an invariant:

Property 1: For any v, if $pweight(v) \neq \infty$, then $pweight(v)$ is the total weight of some path from v_0 to v.

Proof. By induction as usual.

Base: In a start state, the only node whose *pweight* is not ∞ is v_0, and there is a path of total weight 0, namely, the trivial 0-length path, from v_0 to itself.

Inductive step: By inductive hypothesis, $pweight(w)$ before the step is the total weight of some path from v_0 to w. Augmenting this path with one additional edge yields a path to v. ∎

The second of the properties above is a termination property. Invariants don't suffice on their own to prove termination. There are other methods for this, including temporal logic and progress functions. We postpone this. Note that invariants can sometimes be used as *part* of a proof of a termination property.

7 Distributed Algorithms

The techniques of modelling programs as state machines, and reasoning about their behavior using invariants, is very powerful. It extends to parallel programs as well as sequential programs. It even extends to *distributed algorithms*, in which there are many programs running independently, possibly at different speeds, at different nodes of a network.

We give two examples, one simple, one not so simple. We represent each as a single state machine.

7.1 Leader Election

We describe a token-based leader election algorithm, designed to work in a ring of processes.

Imagine a collection of processes numbered $1, \ldots, n$, arranged in a ring (mod n). Each can communicate with its clockwise neighbor, by sending messages over a FIFO message channel. The problem is for exactly one of the processes to be chosen as a leader. We may assume that each starts with a *unique identifier* (*uid*), say, a natural number. Any combination of identifiers can appear in the ring, as long as there are no repeats. This problem originates in ring-based communication networks, where the unique owner of a token is allowed to initiate communication.

In the algorithm we describe here, each process sends its own identifier clockwise to its neighbor, and successive neighbors pass it around the ring. However, each process only passes on identifiers greater than its own. If a process receives an identifier smaller than its own, it simply ignores it. If a process receives its own identifier, it elects itself as the leader. The idea is that only the maximum uid will succeed in getting all the way around the ring.

We represent this algorithm by a state machine. (Actually, when such systems are modelled in practice, each individual process and each individual message channel is usually modelled as a separate state machine, but this requires a more complicated state machine model and formal rules for combining state machine. We avoid these issues here.) The state consists of:

For each i, $1 \leq i \leq n$:
- $val(i)$, i's identifier (initially, all are unique)
- $buffer(i, i+1 \bmod n)$, a finite sequence, initially containing just $val(i)$
- $leader(i)$, a Boolean, initially *false*

The $buffer(i, i+1)$ variable represents the communication buffer for messages from process i to process $i+1$. The transitions are:

$move(i)$
  Precondition:
    $buffer(i-1, i)$ is nonempty
  Effect:
    let $t = head(buffer(i-1, i))$
    remove $head$ of $buffer$
    if $t > val(i)$ then add t to end of $buffer(i, i+1)$
    if $t = val(i)$ then $leader(i) := true$

The precondition of the $move(i)$ action says that the message buffer incoming to i is empty. The effect says that a uid larger than $val(i)$ is passes on to the next process, while a uid equal to $val(i)$ causes i to elect itself as leader.

We would like to prove that exactly one $leader(i)$ flag gets set. For now, however, we'll content ourselves with just showing that *at most* one flag is set. That is, we want to prove that the following is an invariant:

Property 1: At most one $leader(i)$ is $true$.

In order to prove this, we need an auxiliary property: Loosely speaking, it says: "A uid in a buffer is greater than the uids of all processes that it has passed." More formally, we have:

Property 2: If $val(i)$ is in $buffer(j, j+1)$, then $val(i)$ is greater than all of $val(i+1), val(i+2), \ldots, val(j)$.

Property 2 can be proved by induction. The key idea in the inductive step of the proof is that a process only lets the incoming uid pass if it's greater than the process' own uid.

The other part of correctness is showing that some leader is actually eventually elected. This is a termination property, which requires other methods for its proof. However, showing this property turns out to use an auxiliary invariant saying that the maximum uid always appears in some buffer (conservation of maximum uid).

7.2 Alternating Bit Protocol

We finish with a description and proof sketch for the Alternating Bit Protocol, a well-known two-process communication protocol. The protocol achieves reliable FIFO communication over unreliable channels. The unreliable channels may lose or duplicate messages, but are assumed not to reorder them. The protocols allows process 1 to send a sequence of messages from a message alphabet M to process 2.

The algorithm works as follows. Process 1 keeps sending the first message tagged with 1. When process 2 receives this message for the first time, it accepts and outputs it. It sends acknowledgement messages, repeatedly, consisting of the tag value 1. When process 1 receives 1, it stops sending the first message and begins sending the second message, tagged with 0. Process 2, having already accepted the first message tagged with 1, ignores subsequent messages with tag 1, and waits until it sees the first message with tag 0; it accepts this message and outputs it, then acknowledges repeatedly with 0. Process 1 now waits to receive 0, then goes on to the next message with tag 1. In this way, it alternates use of the tags 1 and 0 for successive messages.

We claim that this causes process 2 to output a *prefix* of the sequence of messages that process 1 wishes to send. (The fact that Process 2 actually outputs the entire sequence is a *liveness* claim – liveness properties are a generalization of termination properties. We avoid this argument for now.)

The states consist of:

buf_1, a finite sequence of M, initially arbitrary
$work\text{-}buf_1$, a finite sequence of M, initially equal to buf_1
$tag_1 \in \{0,1\}$, initially 1

buf_2, a finite sequence of M, initially empty
$tag_2 \in \{0,1\}$, initially 0

$channel_{1,2}$, a finite sequence of $M \times \{0,1\}$, initially empty,
$channel_{2,1}$, a finite sequence of $\{0,1\}$, initially empty

Here, buf_1 always contains the complete sequence of messages that process 1 is supposed to send, while $work\text{-}buf_1$ contains the remaining messages to be sent, excluding those that process 1 has already succeeded in sending. The other message buffer, buf_2, contains the outputs. The *tag* variables represent the current tag values that both processes are sending, and *channel* variables represent the low-level messages in transit between the processes. The transitions are:

$send(m,b)_{1,2}$
  Precondition:
    $m = head(work\text{-}buf_1)$
    $b = tag_1$
  Effect:
    add (m,b) to end of $channel_{1,2}$,
      any number ≥ 0 of times

$send(b)_{2,1}$
  Precondition:
    $b = tag_2$
  Effect:
    add b to end of $channel_{2,1}$,
      any number ≥ 0 of times

$receive(m,b)_{1,2}$
  Precondition:
    $(m,b) = head(channel_{1,2})$
  Effect:
    remove *head* of $channel_{1,2}$
    if $b \neq tag_2$ then
      add m to buf_2
      $tag_2 := tag_2 + 1 \bmod 2$

$receive(b)_{2,1}$
  Precondition:
    $b = head(channel_{2,1})$
  Effect:
    remove *head* of $channel_{2,1}$
    if $b = tag_1$ then
      remove head of $work\text{-}buf_1$ (if nonempty)
      $tag_1 := tag_1 + 1 \bmod 2$

Our goal is to show that the following is an invariant:

Property 1: buf_2 is a prefix of buf_1

This requires three auxiliary invariants. For the first of these, we need a definition.

Let *tag-sequence* be the sequence consisting of elements of $channel_{2,1}$, in order, followed by tag_2, followed by the tag components of the elements of $channel_{1,2}$, in order, followed by tag_1.

Property 2: *tag-sequence* consists of one of the following:

1. All 0's.
2. All 1's.
3. A positive number of 0's followed by a positive number of 1's.
4. A positive number of 1's followed by a positive number of 0's.

What is being ruled out by these four cases is the situation where the sequence contains more than one switch of tag value.

The fact that Property 2 is an invariant can be proved easily by induction. We also need:

Property 3: If (m, tag_1) is in $channel_{1,2}$ then $m = head(work\text{-}buf_1)$.

Proof. (That Property 3 is an invariant)

By induction, using Property 2.

Base: Obvious, since no message is in the channel initially.

Inductive step: It is easy to see that the property is preserved by $send_{m,b}$, which adds new messages to $channel_{1,2}$. The only other case that could cause a problem is $receive(b)_{2,1}$, which could cause tag_1 to change when there is another message already in $channel_{1,2}$ with the same tag. But this can't happen, by Property 2 applied before the step – since the incoming tag g must be equal to tag_1 in this case, all the tags in *tag-sequence* must be the same. ■

Finally, we need that the following two-part property is an invariant:

Property 4:

1. If $tag_1 \neq tag_2$ then $buf_2 \cdot work\text{-}buf_1 = buf_1$.
2. If $tag_1 = tag_2$ then $work\text{-}buf_1$ and buf_2 are nonempty, $head(work\text{-}buf_1) = last(buf_2)$, and $buf_2 \cdot tail(work\text{-}buf_1) = buf_1$.

Here, *tail* means all but the head of the sequence.

Property 4, part 2, easily implies the goal Property 1. It also implies that $work\text{-}buf_2$ is always nonempty when $receive(b)_{2,1}$ occurs with equal tags; therefore, the parenthetical check in the code always works out to be true.

Proof. (That Property 4 is an invariant)

By induction. Base: In an initial state, the tags are unequal, $work\text{-}buf_1 = buf_1$ and buf_2 is empty. This suffices to show part 1. part 2 is vacuous.

Inductive step: When a *send* occurs, the tags and buffers are unchanged, so the truth of the invariants must be preserved. It remains to consider *receive* events.

$receive(m,b)_{1,2}$:
If $b = tag_2$, nothing happens, so the invariants are preserved. So suppose that $b \neq tag_2$. Then Property 2 implies that $b = tag_1$, and then Property 3 implies that m is the first message on $work\text{-}buf_1$. The effect of the transition is to change tag_2 to make it equal to tag_1, and to replicate the first element of $work\text{-}buf_1$ at the end of buf_2.

The inductive hypothesis implies that, before the step, $buf_2 \cdot work\text{-}buf_1 = buf_1$. The changes caused by the step imply that, after the step, $tag_1 = tag_2$, $work\text{-}buf_1$ and buf_2 are nonempty, $head(work\text{-}buf_1) = last(buf_2)$, and $buf_2 \cdot tail(work\text{-}buf_1) = buf_1$. This is as needed.

$receive(b)_{2,1}$:
The argument is similar to the one for $receive(m,b)_{1,2}$. If $b \neq tag_1$, nothing happens so the invariants are preserved. So suppose that $b = tag_1$. Then Property 2 implies that $b = tag_2$, and the step changes tag_1 to make it unequal to tag_2. The step also removes the first element of $work\text{-}buf_1$. The inductive hypothesis implies that, before the step, $work\text{-}buf_1$ and buf_2 are nonempty, $head(work\text{-}buf_1) = last(buf_2)$, and $buf_2 \cdot tail(work\text{-}buf_1) = buf_1$. The changes caused by the step imply that, after the step, $tag_1 \neq tag_2$ and $buf_2 \cdot work\text{-}buf_1 = buf_1$. This is as needed. ■

8 Liveness

There is no room in this course to discuss these issues in any detail. We just give a quick overview.

A safety property, informally speaking, says that some particular thing (presumably, an undesirable event) never occurs. A liveness property, in contrast, talks about things that must eventually occur (or must occur periodically). Termination is an important special case of a liveness property.

Invariants are useful for proving safety properties, but are not in themselves sufficient for proving liveness properties. They can sometimes be used to help in proving liveness, though – recall the conservation of max-uid invariant in the leader election example.

Typical tools for proving liveness properties include a formal logic known as *temporal logic* and a device known as a *progress function.*

Temporal logic, originally developed by Pnueli (the latest Turing award winner), makes statements about an execution α together with a pointer i to a state s_i in the execution. It involves operators $\Box$ and $\Diamond$ – $\Box P$ means that property P is true in all states starting from s_i, while $\Diamond P$ means that the property P is true in some state starting from s_i (i.e., that P is "eventually true", after s_i). These can be combined into formulas; for instance, $\Box \Diamond P$ means that, if α is infinite, then P is true in infinitely many of the states of α. There are formal rules for manipulating statements involving these symbols, which can help in proving liveness properties.

A progress function is a measure taken from a well-founded set – usually a natural number or a tuple of natural numbers – associated with each state. Every step toward some goal should decrease the value of the progress function (tuples would be ordered lexicographically). Thus, if steps continue to occur (we must show that there is always a step whose precondition holds), the progress function's value should continue to decrease, unless the goal is reached. But it cannot decrease indefinitely, because the set is well-founded. This shows that the goal must eventually be reached.

In Euclid's algorithm, we can regard the first input argument (or the sum of the two arguments, or the tuple consisting of the two arguments) as a progress function.

9 Simulation Relations

Another important technique involves proving correctness of state machines in a hierarchical manner. First, an abstract description of an algorithm as a state machine is proved correct. Then, a more detailed description is related formally to the abstract description. This is done in such a way that correctness properties of the abstract algorithm are carried over automatically to the more detailed algorithm. This method is extremely practical for system verification – for example, the Principles of Computer Systems course, 6.026, uses this method throughout. The method is especially valuable for safety properties, though it is of help also for liveness.

10 Exercises

1. Consider a state machine whose states consist of values for the following variables, all in $\mathbb{N}$:

   x, y: initially arbitrary positive naturals
   a: initially 1
   r, s: initially equal to x and y, respectively

   Its transitions are of the following two kinds:

   even
   Precondition:
   s is even
   $s > 0$
   Effect:
   $r := r^2$
   $s := \frac{s}{2}$

   odd
   Precondition:
   s is odd
   Effect:
   $a := a \times r$
   $r := r^2$
   $s := \frac{s-1}{2}$

Use induction on the length of an execution to prove the following invariant of this state machine:

$$\text{If } s = 0 \text{ then } a = x^y.$$

(*Hint: You will need to strengthen this invariant in order to prove it by induction.*)

2. Consider a state machine whose state consists of:

    V, a set of vertices, initially containing one vertex.
    E, a collection of two-element subsets of V, initially empty
    $n \in \mathbb{N}$, initially 1.

    The transitions are:

    expand(v, w, x)
    Precondition:
    $v \in V$
    there is at most one set containing v in E
    $w, x \notin V$
    Effect:
    $V := V \cup \{w, x\}$
    $E := E \cup \{\{v, w\}, \{v, x\}\}$
    $n := n + 2$

    Prove (in order) that the following properties are invariants:

    (a) $|V| = n$
    (b) (V, E) is a tree
    (c) If $|V| \geq 2$ then (V, E) has $\frac{n-1}{2}$ internal nodes.

3. Develop a state machine for a very general Minimum Spanning Tree algorithm. The state contains an undirected weighted graph, and a current forest which is a subgraph of the given graph. A transition is allowed to add in another edge, defined to be the smallest outgoing edge from some arbitrary component of the forest.

    (a) Give a formal definition for the state machine.
    (b) Prove that the following is an invariant of your state machine: The current forest is always a subgraph of a minimum spanning tree of the given graph.