Reasoning About State Machines

Garlan & Wing Handout 7 22 September 1997

Given a state machine model of a system, we can do some formal reasoning about properties of the model. It's important to remember that we are proving some property about the *model of the system*, not the *system* itself. If the model is "incorrect" then we may not be able to prove anything useful about the system. Worse, if the model is "incorrect" then we may be able to prove something that has no correspondence to the real system. However, we hope that we've modeled our system properly so that whatever we prove about our model is true of the system being modeled.

But then, why not just reason about the system itself directly? One reason is that it's often impossible to reason about the system itself because it's too large, too complex, or too unwieldy. Another is that we may be interested in one aspect of the system and want to abstract from the irrelevant aspects. Another is that we may not actually have a real system; our model could simply be a high-level design of a system we might build and we want to do some reasoning about our design before spending the dollars building the real thing. Another is that it may be impossible to get our hands on the system (maybe it's proprietary). Another is that it may be impossible for us to run the system to check for the property because of its safety-critical side-effects (like setting off a bomb). So, in some cases, the best we can do is reason about a model of the system, and not the system itself.

In this handout I'll discuss a few kinds of properties we might want to reason about a state machine model.[1] The most important of these is *invariant properties*, properties that are true of every reachable state in the system.

1 Invariants

An *invariant* is a predicate that is true in all states. In the context of state machines, we usually care that an invariant is true in all *reachable* states. The statement of an invariant, θ, in full generality looks like:

$$\forall\, e : \mathit{executions} \bullet \forall\, s : S \bullet s \,\mathbf{in}\, e \Rightarrow \theta(s)$$

where θ is a predicate over variables in s and **in** is a predicate that says whether a state is in an execution. Normally the universal quantification over all executions and the condition that s be in e is omitted (it is understood):

$$\forall\, s : S \bullet \theta(s)$$

Sometimes we also omit the universal quantification over s as well because it's also understood:

$$\theta(s)$$

For example, here is an invariant for the Counter example given in State Machine Variations handout:

$$s(x) \geq 0$$

which says that in all states, x's value is greater than or equal to 0. We know this is true because initially x's value is 0 and because the *inc* action always increases x's value by 1. Since *inc* is Counter's only action, there's no other way to change x.

[1] In this handout, I'm going to be even less precise in my notation and use of formalism for the sake of clarity of presentation.

1.1 Proving an Invariant

How do we show that a predicate is an invariant? There are lots of techniques. If the state machine is finite, you can do an exhaustive case analysis and show that it holds for every state. This technique is fine if there are a small number of states or if you have a tool called a *model checker* handy (you will in your Analysis core course next semester).

If your state machine is infinite, you must resort to something else. I'm going to sketch out three techniques. They can all also be used if your state machine is finite. Technique C is the one most often used in practice.

A. Use induction over states in executions.

When you have to reason about an infinite domain, the technique that should spring to mind is *induction*. Induction is especially appropriate when there is a natural, often recursive, structure to your domain. Since what we want to prove is that a property is true of every state of every execution of the state machine, then I can induct over the states in the sequence of states of every execution. Recall that an execution looks like:

$$\langle s_0, a_1, s_1, a_2, \ldots, s_{i-1}, a_i, s_i, \ldots \rangle$$

Then to prove a property, θ, is invariant requires that for every execution:

1. Base Case: I show it holds in the initial state s_0, and
2. Inductive Step: I assume it holds in state s_{i-1} and show that it holds in state s_i.

B. Show that your state space predicate implies the invariant.

A second technique is that if you're lucky the predicate that you've used to define your state space is stronger than the invariant you're trying to prove. So regardless of whether a state is reachable or not, you can prove the invariant holds:

$$P \Rightarrow \theta$$

where P is the predicate describing your set of states. If it's true of every state, then certainly it's true of every reachable state.

For example, recall in the Counter example, the predicate P is simply $s(x) \geq 0$. Hence we can trivially show the invariant property holds:

$$s(x) \geq 0 \Rightarrow s(x) \geq 0$$

C. Use a proof rule using pre-/post-condition specifications.

Technique A requires that we reason in terms of first principles— using structural induction (over states in executions)—which can sometimes be cumbersome. And, usually, of course, we are not so lucky as in Technique B. Technique C is an alternative inductive proof strategy that is usually more manageable than Technique A: we do a case analysis of all actions, which indirectly proves the same thing as in Technique A.

At the bottom of p. 1 when I argued that the Counter's invariant holds, we need to show that the invariant holds in each initial state and then for each action[2] show that if the invariant holds in its pre-state, it holds in the post-state. In general, we have a proof rule that looks like:

[2] Here again, is another good reason to have only a *finite* set of actions.

$$\frac{\forall\, s : I \bullet \theta(s) \qquad \forall\, a : A \bullet \forall\, s, s' : S \bullet (s, a, s') \in \delta \Rightarrow ((\Phi(s) \wedge \theta(s) \wedge \Psi(s, s')) \Rightarrow \theta(s'))}{\forall\, s : S \bullet \theta(s)}$$

where I is the set of initial states and A is the set of actions. Φ and Ψ are the pre- and post-conditions of a, respectively. Or, said in English:

1. Show that θ is true for each initial state.
2. For each action,
   - assume
     - the pre-condition Φ holds in the pre-state,
     - the invariant θ holds in the pre-state, and
     - the post-condition Ψ holds in the pre- and post-states, then
   - show
     - θ holds in the post-state.
3. Conclude that θ is an invariant.

The two main proof steps are sometimes called (1) *establishing the invariant* (true in initial states) and (2) *preserving the invariant* (assuming it's true in a pre-state and showing that each action's the post-state preserves it).

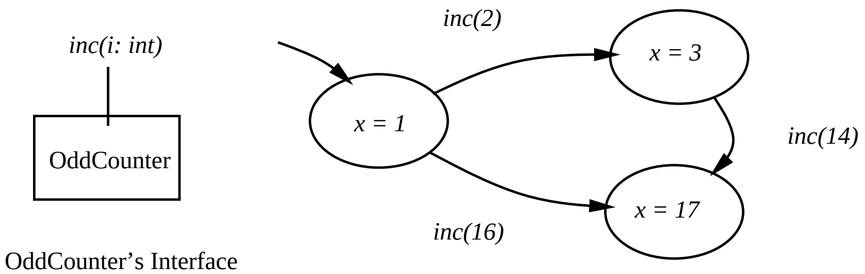
What's the rationale for this proof rule? First, notice I care only about reachable states (that's why the δ appears above). Second, consider any execution of my state machine:

$$\langle s_0, a_1, s_1, a_2, \ldots, s_{i-1}, a_i, s_i, \ldots \rangle$$

As in Technique A I need to make sure that θ holds in s_0 (establishing the invariant). Also, for any pair of successive states, (s_{i-1}, s_i), in the execution, if I assume θ holds in s_{i-1} then I need to show it holds in s_i. Since the only way I can get from any s_{i-1} to the next state s_i is by one of the actions a in A, then if I show the invariant if preserved for each a, I've shown for each reachable state the invariant holds.

1.2 OddCounter

Let OddCounter be the state machine:



OddCounter's Interface

Part of OddCounter's State Transition Diagram

More precisely,

OddCounter = (
$\{s : \{x\} \rightarrow int\}$,
$\{s : \{x\} \rightarrow int \mid s(x) = 1\}$,
$\{inc(i : int)\}$,
$\delta =$

inc(i: int)
  pre i is even
  post $x' = x + i$

).

The invariant I want to prove is that OddCounter's state always holds an odd integer:

$$\theta \equiv x \text{ is odd.}$$

Intuitively we see this is true because

1. It's true of the initial state (1 is an odd integer).
2. For the action *inc*:
   - I assume:
     - i is even (i.e., the pre-condition holds in the pre-state),
     - x is odd (i.e., the invariant holds in the pre-state), and
     - $x' = x + i$ (i.e., the post-condition holds in the pre- and post-states)
   - I need to show that x' is odd.
     - This is true since from facts about numbers I know an odd number (x) plus an even number (i) is always odd.

1.3 Fat Sets

This example has two purposes. One is to give another example of an invariant. The other is to hint at how state machines are appropriate models of abstract data types, as you might implement in your favorite object-oriented language.

Here is a description of a FatSet abstract data type modeled as a state machine:

- $S = \{s : \{t\} \rightarrow set[int]\}$. The state variable, t, maps to a set of integers.
- $I = \{s : \{t\} \rightarrow set[int] \mid \#s(t) = 1\}$. The set of initial states is the set of states that map t to a singleton set. Notice that there are an infinite number of initial states.
- A = { *union(u: set[int])/ok(), card()/ok(int)* }
- δ =

union(u: set[int])/ok()
  pre $u \neq \emptyset$
  post $t' = t \cup u$

card()/ok(int)
  pre *true*
  post $result = \#t \wedge t' = t$

Suppose I want to show the property that the size of the FatSet t is always greater than or equal to 1:

$$\theta \equiv \#s(t) \geq 1$$

Here's an informal proof:

1. It's true of all initial states since the size of all singleton sets is 1.
2. We need to show the invariant is preserved for each of *union* and *card*.
   (a) (*union*). Assume
   - u is nonempty (i.e., the pre-condition holds in the pre-state),
   - t's size is ≥ 1 (i.e., the invariant holds in the pre-state), and
   - $t' = t \cup u$ (i.e., the post-condition holds in the pre- and post-states)

   Then I need to show that the size of t' is ≥ 1. This is true since taking the union of two non-empty sets (t and u) is a non-empty set, whose size is ≥ 1.

   (b) (*card*). Assume
   - t's size is ≥ 1 (i.e., the invariant holds in the pre-state), and
   - $result = \#t \wedge t' = t$ (i.e., the post-condition holds in the pre- and post-states)

   Then I need to show that the size of t' is ≥ 1. This is true since $t' = t$ and t's size is ≥ 1.

Notice how awful it would be if I had to write out these proof steps in gory detail!

Two points about this example:

- You should notice that the only interesting part of the proof above was for *union*, the only action that changes the value of any state variable. An action that changes the state of some state variable is called a *mutator*. I didn't really have anything to prove for *card* because the action does not change the state of any state variable. I'll call such an action a *non-mutator*. In practice, we need to show only for mutators that the invariant is preserved because non-mutators by definition cannot change state. So if the invariant holds before the non-mutator is called (pre-state); then it holds afterwards (post-state).
- Second I deliberately chose a value space for t that includes "unreachable values." In particular, t can never be the empty set because it starts out non-empty and always remains non-empty. But, remember, I need to show the invariant holds of only reachable states.

Now, suppose I added a *delete* action to my FatSet example. Let *delete* have the following behavior:

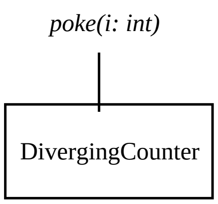
delete(i:int)/ok()
  pre $t \neq \emptyset$
  post $t' = t \setminus \{i\}$

Then my invariant would no longer hold because if *delete* were called in any state where $t = \{i\}$ (where i is the argument to *delete*) then the size of t' would be 0.

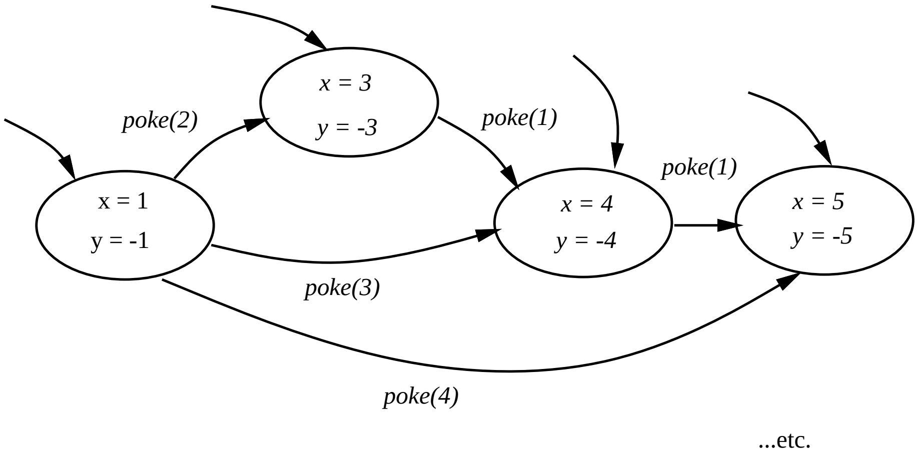
1.4 Diverging Counter

This example has two purposes. One is to give another example of an invariant. The other is to give an example of a state machine with more than one state variable. The invariant property captures an invariant relationship that we want to maintain between these two state variables.

Here's a two-integer counter with state variables, x and y, whose values get further and further apart as you *poke* the machine.



DivergingCounter's Interface



Part of DivergingCounter's State Transition Diagram

Here's a description of the DivergingCounter:

DivergingCounter = (
$\{s : \{x, y\} \rightarrow int\}$,
$\{s : \{x, y\} \rightarrow int \mid s(x) = -s(y)\}$,
$\{poke(i : int)\}$,
$\delta =$

poke(i: int)
pre $i > 0$
post $x' = x + i \wedge y' = y - i$

).

Notice that all the states drawn above are legitimate initial states. So is the state where x and y are both initialized to 0.

The invariant maintained by DivergingCounter is:

$$\theta \equiv s(x) + s(y) = 0$$

Can you prove it?

1.5 Metacomment

In invariants it's awkward writing $s(x)$ and $s(y)$ all the time since s is universally quantified. So more typically you'll see invariants expressed directly in terms of the state variables. For the DivergingCounter, you'd more likely see its invariant in this more readable format:

$$\theta \equiv x + y = 0$$

We use this syntactic sugar in the pre- and post-conditions already.

2 Constraints

An invariant is supposed to be true of every *state* of every execution of a state machine. You might ask is there a corresponding notion for *state transitions*? The answer is "yes," though because it is not as commonly discussed in the literature, there is no common term for such a property. Some people might simply call it another kind of invariant, one over state transitions rather than states. To avoid confusion with state invariants, I'm going to call it a *constraint*. This word is not standard; also, others may use "constraint" to mean something different.

Consider any execution of a state machine:

$$\langle s_0, a_1, s_1, a_2, \ldots, s_{i-1}, a_i, s_i, a_{i+1}, s_{i+1}, \ldots, s_{j-1}, a_j, s_j, \ldots \rangle$$

A *constraint* is a predicate that is true in all pairs of states, s_i and s_j, in every execution, where s_j follows s_i, but need not immediately follow it (s_j does not have to be s_{i+1}).

The statement of a constraint, χ, in full generality looks like:

$$\forall\, e : \mathit{executions} \bullet \forall\, s_i, s_j : S \bullet (s_i \,\mathbf{in}\, e \wedge s_j \,\mathbf{in}\, e \wedge i < j) \Rightarrow \chi(s_i, s_j)$$

As for statements of invariants, we omit (because it's understood) the universal quantification over executions and states; the condition about s_i and s_j both being states in the execution; and the condition that s_i precedes s_j in the execution.

A constraint that holds for the Counter example is that its value always strictly increases:

$$s'(x) > s(x)$$

Does this constraint (appropriately restated) hold for SimpleCounter? BigCounter (careful!)? FatCounter? RandomCounter?

2.1 Proving Constraints

If the constraint we are interested in proving is a "transitive" property (if it holds for s_i and s_{i+1} and for s_{i+1} and s_{i+2} then it holds for s_i and s_{i+2}) then we can use the following proof rule to show the constraint holds of any execution of the state machine:

$$\frac{\forall\, a : A \bullet \forall\, s, s' : S \bullet (s, a, s') \in \delta \Rightarrow (\Phi(s) \wedge \Psi(s, s') \Rightarrow \chi(s, s'))}{\chi(s_i, s_j)}$$

where A is the set of actions and s_i and s_j are quantified and qualified as described above. Or, in English:

1. For each action $a \in A$,
   - assume
     - Φ holds in the pre-state,
     - Ψ holds in the pre/post-states, and
   - show
     - χ holds in the pre/post-states.
2. Conclude that χ holds of all pairs of states in all executions.

What's the rationale for this proof rule? First, again I care only about reachable states. Second, consider any execution of my state machine:

$$\langle s_0, a_1, s_1, a_2, \ldots, s_{i-1}, a_i, s_i, a_{i+1}, s_{i+1}, \ldots, s_{j-1}, a_j, s_j, \ldots \rangle$$

If I can show that χ holds over any pair of successive states, (s_i, s_{i+1}), i.e., every single state transition, then surely it holds over any pair of states, (s_i, s_j), where $i < j$. To show that it holds in any pair of successive states, I need only consider every possible action, which is the only way I can get from s_i to s_{i+1}. For each action, I need to make sure that the conjunction of its pre- and post-condition predicates imply the constraint.

2.2 Fat Sets Again

Here is a constraint for my FatSet example:

$$\chi \equiv \forall\, x : int \bullet x \in s_i(t) \Rightarrow x \in s_j(t)$$

where s_i and s_j are implicitly defined and qualified as usual. This says that once an integer gets added to my set t it never disappears. We know this constraint holds because there is no way to delete elements from the set.

Notice that saying that the cardinality of t always strictly increases:

$$\text{WRONG! } \chi \equiv \#s_i(t) > \#s_j(t)$$

is not a constraint for FatSet. It does not hold since taking the union of two sets may not necessarily increase the size of either.

2.3 MaxCounter

Constraints are useful for stating succinctly when things do not change in value. Consider the following MaxCounter machine whose state variable x can never exceed the value of the other state variable *max*. *Max* is initialized to 15 and its value never changes.

MaxCounter = (
$\{s : \{x, max\} \rightarrow int\}$,
$\{s : \{x, max\} \rightarrow int \mid s(x) = 0 \wedge s(max) = 15\}$,
$\{inc(i : int)\}$,
$\delta =$

inc(i: int)
pre $x + i \leq max$
post $x' = x + i \wedge max' = max$

).

It's trivial to show the following constraint:

$$\chi \equiv s_i(max) = s_j(max)$$

This kind of example may look simplistic but it generalizes to any system where you want to ensure that some state variable never changes. When we have a huge state space (as is more typical of software systems than in my zillions of simple counter examples), very often we are careful to state how some state variable changes but forget to say what state variables do not change. Constraints are a nice way to describe those properties.

3 Other Properties of State Machines

The kinds of properties you have seen so far are sometimes called *safety* properties, properties that say that "nothing bad happens." Another class of properties that people often discuss are called *liveness* properties, properties that say that "something good eventually happens." For a sequential system, computing the correct answer is an example of a safety property and termination of a program is an example of a liveness property (this program "eventually" terminates). For a concurrent system, deadlock freedom and mutual exclusion are examples of safety properties; for a distributed system, the property that a message sent is eventually received is an example of a liveness property.

For concurrent and distributed systems, there are many interesting liveness properties so we will defer discussing them until later in the course.