Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 8 September 15, 1993

Abstraction Functions and Rep Invariants

This handout discusses how abstraction functions and representation invariants can be used to show the correctness of an implementation of a data abstraction. It uses the SimpleMemory and WBCache modules from Handout 5 as examples.

1 State Machine Models

We will abstract from the text of a module with internal state to view it as a kind of automaton, or state machine. The states of the automaton are determined by the global variables of the module, and the transitions of the automaton correspond to the atomic actions. For sequential programs such as the examples in Handout 5, the atomic actions are simply the executions of the routines. (We will see in a couple of weeks how concurrency complicates things.) From the viewpoint of a client of a module, we can describe an execution of a routine as a pair consisting of the invocation (routine name and argument values) and the response (result, either normal or exceptional). Some of the states and transitions of the state machine denoted by the SimpleMemory module are shown in Figure 1, and similarly in Figure 2 for the WBCache module.

2 Correctness

Since the state domain of an implementation module is typically different from that for its specification module, we cannot use the simple "subset" definition for correctness that we used for procedures. Instead, we need to use a more abstract view of the modules that hides the internal states. The model we will use is to view each module as denoting a set of *traces*, where each trace represents one possible execution of the module. A trace is simply a sequence of (invocation, response) pairs, each pair representing a single execution of one of the routines provided by the module.

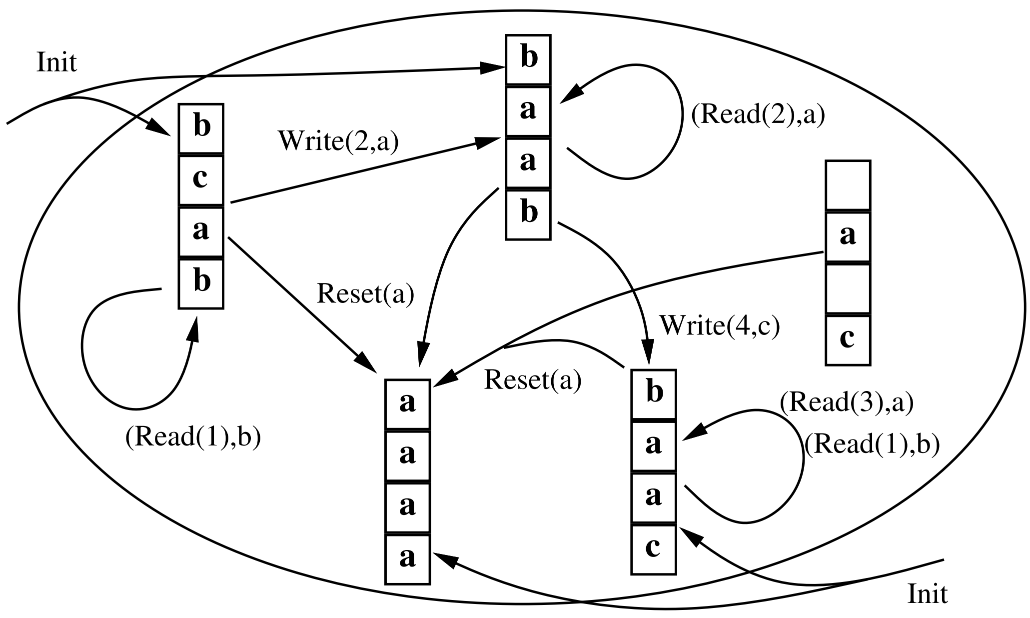

Figure 1: SimpleMemory state machine

For example, the set of traces denoted by the SimpleMemory module includes the following trace:

```
(Reset(d),RET)
(Read(a1),d)
(Write(a2,x),RET)
(Read(a2),x)
```

However, the following trace is not included:

```
(Reset(d),RET)
(Read(a1),x)
(Write(a2,x),RET)
(Read(a2),d)
```

In general, a trace is included if every Read operation returns the last value written by a Write or Reset operation. (We are ignoring Swap operations in these examples.)

We define correctness for modules with state as follows: Let S_I be the set of traces denoted by an implementation module, and let S_S be the set of traces denoted by its specification. Then the implementation is *correct* if $S_I \subseteq S_S$. (Note: this definition will change slightly when we introduce concurrency.)

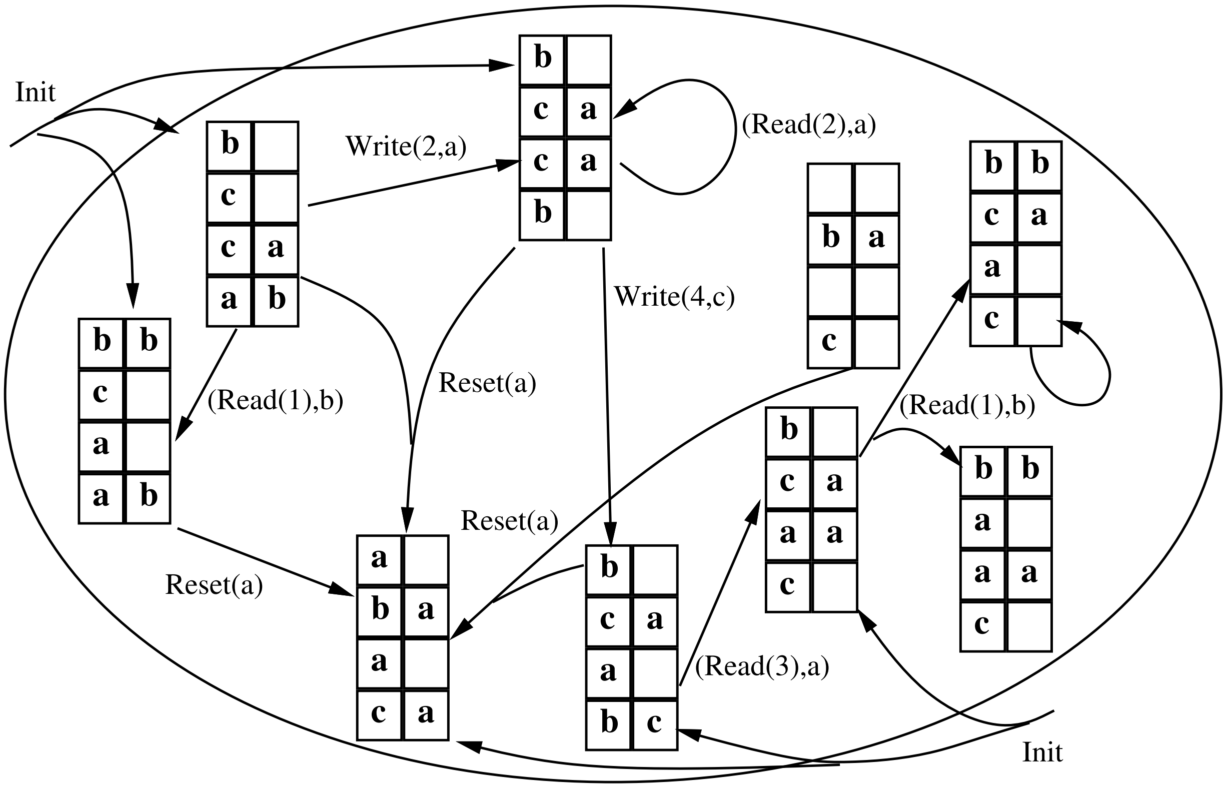

Figure 2: WBCache state machine

3 Proving Correctness

The basic definition for correctness given in the previous section is quite simple. Showing that an implementation is correct requires a little more mechanism. In particular, we will use *abstraction functions* and *representation invariants*.

An abstraction function maps a state of the implementation to a state of the specification. For example, a state of the WBCache module gets mapped to a state of the SimpleMemory module. The abstraction function explains how to interpret each state of the implementation as a state of the specification. It solves the problem of the implementation and the specification having different state domains. For example, Figure 3 illustrates the abstraction function from WBCache to SimpleMemory. (Handout 5 contains a definition of this abstraction function in Spec.)

You might think that the abstraction function should map the other way, explaining how to represent each state of the specification. This is usually not a good idea, in large part because there is often more than one way of representing each state of the specification. For example, for the WBCache implementation of SimpleMemory, if an address is in the cache, the value stored for it in memory doesn't matter. Similarly, there are choices about which addresses appear in the cache. Another way of saying this is that many states of the implementation represent the same state of the specification; in other words, the abstraction function is many-to-one.

To use an abstraction function to prove implementations correct, we must show the following two conditions (where AF is the abstraction function):

1. For every initial state s of the implementation, AF(s) is an initial state of the specification.

2. For every transition p from s_1 to s_2 in the implementation, there must be a transition p from AF(s_1) to AF(s_2) in the specification.

(Recall that each transition is represented by an (invocation, response) pair.) The second part of the rule above is represented by the diagram in Figure 4. The dashed arrows represent the abstraction function, and the solid arrows represent the transitions; if the lower (double) solid arrow exists in the implementation, the upper (single) solid arrow must exist in the specification. The diagram is sometimes called a "commutative diagram" because if you start at the lower left and follow the arrows, you will end up in the same state regardless of which way you go.

For the WBCache implementation of SimpleMemory, the abstraction function is enough to show correctness. For some implementations, however, we need more; in particular, we need to use the fact that some states of the implementation might not be reachable during a computation. For example, consider the HashMemory implementation of SimpleMemory in Handout 5. In some of the possible states of the global variables, the data for a given address could be in the wrong bucket, in which case the Read operation will give the wrong answer. However, in every reachable state (one that can be obtained from the initial state by executing operations), data is always in the bucket given by the hash function. Thus, this problem can never arise.

To deal with this kind of situation, we first define an invariant, sometimes called a rep invariant, on the state of the implementation. One possible invariant for HashMemory is given in Handout 5. The invariant must have the property that it is true of every reachable state; this can be proved inductively as follows:

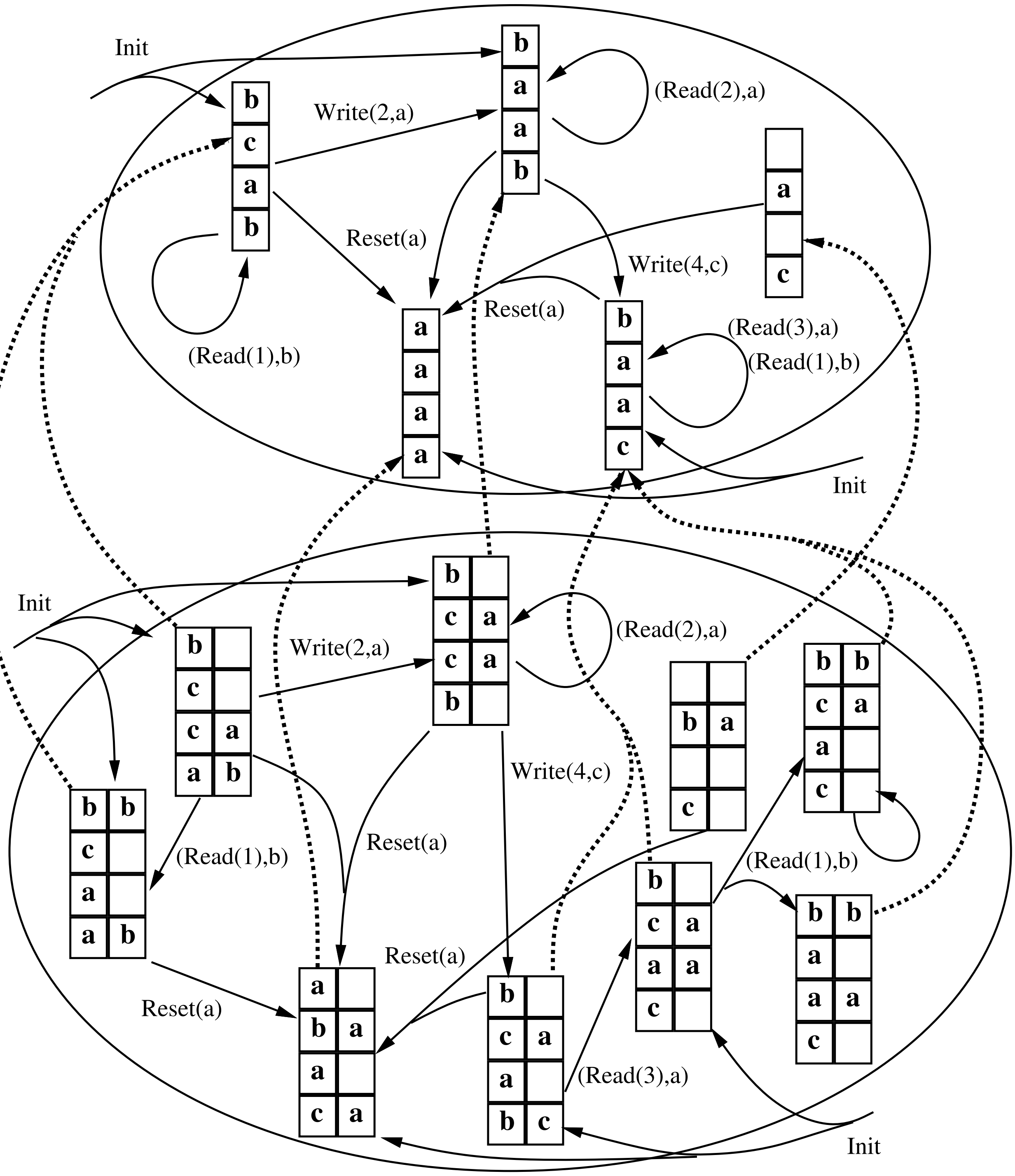

Figure 3: Abstraction function for WBCache

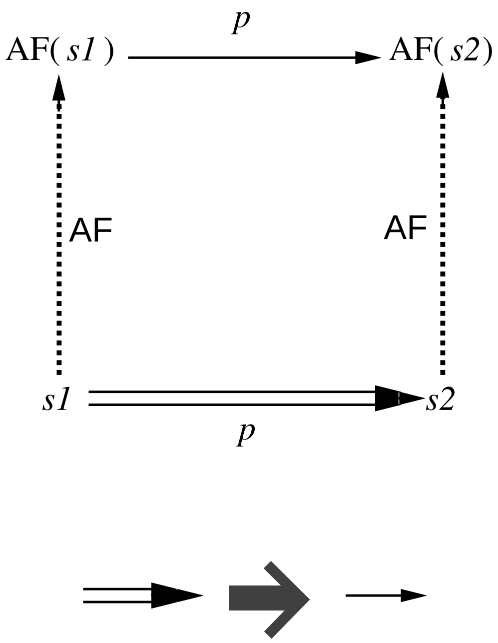

Figure 4: Commutative diagram for correctness

1. Show that every initial state satisfies the invariant.

2. Show that if there is a transition from state s_1 to state s_2 where s_1 satisfies the invariant, then s_2 also satisfies the invariant.

Next, we modify the conditions on the abstraction function to consider only transitions that start in states that satisfy the invariant; since all reachable states satisfy the invariant, we are still considering every transition that can ever happen. The revised conditions are:

1. For every initial state s of the implementation, AF(s) is an initial state of the specification.

2. For every transition labeled p from s_1 to s_2 in the implementation where s_1 satisfies the rep invariant, there must be a transition labeled p from AF(s_1) to AF(s_2) in the specification.

The rep invariant essentially serves to restrict the domain of the abstraction function.