6. Abstraction Functions and Invariants

This handout describes the main techniques used to prove correctness of state machines: abstraction functions and invariant assertions. We demonstrate the use of these techniques for some of the `Memory` examples from handout 5.

Throughout this handout, we consider modules all of whose externally invocable procedures are `APROC`s. We assume that the body of each such procedure is executed all at once. Also, we do not consider procedures that modify global variables declared outside the module under consideration.

Modules as state machines

Our methods apply to an arbitrary state machine or automaton. In this course, however, we use a Spec module to define a state machine. Each state of the automaton designates values for all the variables declared in the module. The initial states of the automaton consist of initial values assigned to all the module's variables by the Spec code. The transitions of the automaton correspond to the invocations of `APROC`s together with their result values.

An *execution fragment* of a state machine is a sequence of the form s_0, π_1, s_1, π_2, …, where each s is a state, each π is a label for a transition (an invocation of a procedure), and each consecutive (s_i, π_{i+1}, s_{i+1}) triple follows the rules specified by the Spec code. (We do not define these rules here—wait for the lecture on atomic semantics.) An *execution* is an execution fragment that begins in an initial state.

The π_i are labels for the transitions; we often call them *actions*. When the state machine is written in Spec, each transition is generated by some atomic command, and we can use some unambiguous identification of the command for the action. At the moment we are studying sequential Spec, in which every transition is the invocation of an exported atomic procedure. We use the name of the procedure, the arguments, and the results as the label.

Figure 1 shows some of the states and transitions of the state machine for the `Memory` module with `A = IN 1 .. 4`, and Figure 2 does likewise for the `WBCache` module with `Csize = 2`. The arrow for each transition is labeled by its π_i, that is, by the procedure name, arguments, and result.

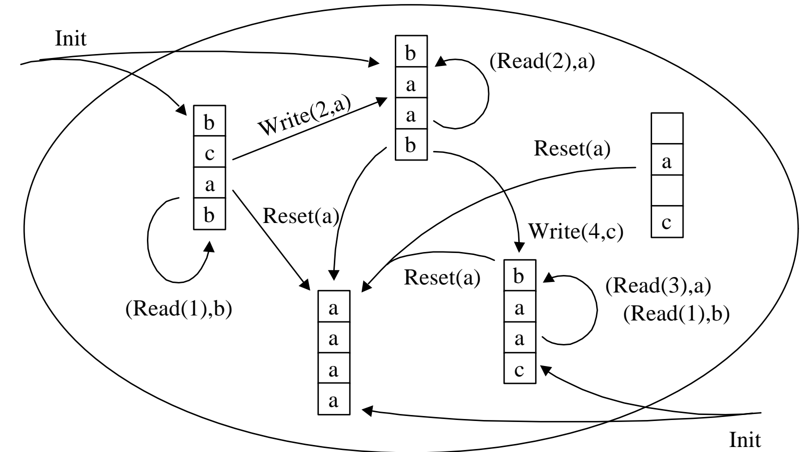

Figure 1: Part of the `Memory` state machine

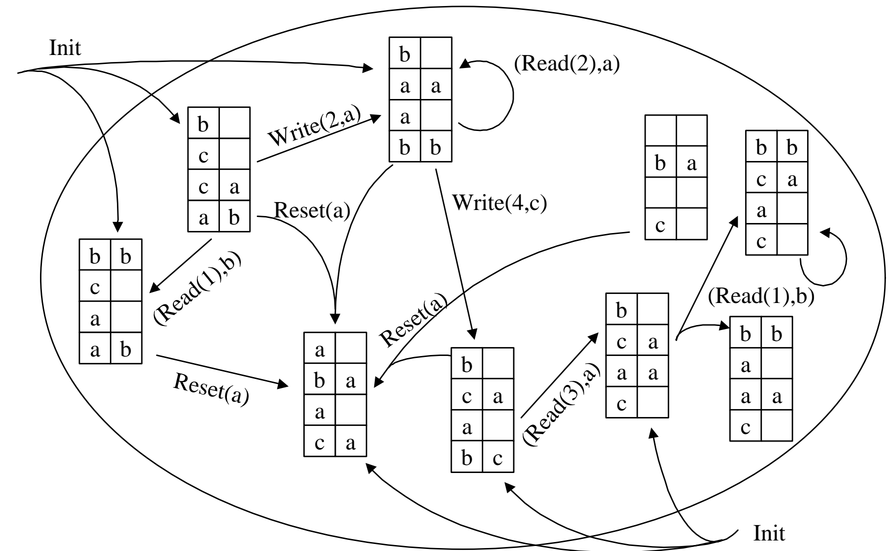

Figure 2: Part of the `WBCache` state machine

External behavior

Usually, a client of a module is not interested in all aspects of its execution, but only in some kind of external behavior. Here, we formalize the external behavior as a set of *traces*. That is, from an execution (or execution fragment) of a module, we discard both the states and the internal actions, and extract the *trace*. This is the sequence of labels π_i for external actions (that is, invocations of exported routines) that occur in the execution (or fragment). Then the external behavior of the module is the set of traces that are obtained from all of its executions.

It's important to realize that in going from the execution to the trace we are discarding a great deal of information. First, we discard all the states, keeping only the actions or labels. Second, we discard all the internal actions, keeping only the external ones. Thus the only information we keep in the trace is the behavior of the state machine at its external interface. This is appropriate, since we want to study state machines that have the same behavior at the external interface; we shall see shortly exactly what we main by 'the same' here. Two machines can have the same traces even though they have very different state spaces.

In the sequential Spec that we are studying now, a module only makes a transition when an exported routine is invoked, so all the transitions appear in the trace. Later, however, we will introduce modules with internal transitions, and then the distinction between the executions and the external behavior will be important.

For example, the set of traces generated by the `Memory` module includes the following trace:

```
(Reset(d),)
(Read(a1),d)
(Write(a2,d'))
(Read(a2),d')
```

However, the following trace is not included if `d # d'`:

```
(Reset(d))
(Read(a1),d')          should have returned d
(Write(a2,d'))
(Read(a2),d)           should have returned d'
```

In general, a trace is included in the external behavior of `Memory` if every `Read(a)` or `Swap(a, d)` operation returns the last value written to `a` by a `Write`, `Reset` or `Swap` operation, or returned by a `Read` operation; if there is no such previous operation, then `Read(a)` or `Swap(a, d)` returns an arbitrary value.

Implements relation

In order to understand what it means for one state machine to implement another one, it is helpful to begin by considering what it means for one atomic procedure to implement another. The meaning of an atomic procedure is a relation between an initial state just before the procedure starts (sometimes called a 'pre-state') and a final state just after the procedure has finished (sometimes called a 'post-state'). This is often called an 'input-output relation'. For example, the relation defined by a square-root procedure is that the post-state is the same as the pre-state, except that the square of the procedure result is close enough to the argument. This meaning makes sense for an arbitrary atomic procedure, not just for one in a trace.

We say that procedure *P* implements spec *S* if the relation defined by *P* (considered as a set of ordered pairs of states) is a subset of the relation defined by *S*. This means that *P* never does anything that S couldn't do. However, P doesn't have to do everything that S can do. An implementation of square root is probably deterministic and always returns the same result for a given argument. Even though the spec allows several results (all the ones that are within the specified tolerance), we don't require an implementation to be able to produce all of them; instead we are satisfied with one.

Actually this is not enough. The definition we have given allows *P*'s relation to be empty, that is, it allows *P* not to terminate. This is usually called 'partial correctness'. In addition, we usually want to require that *P*'s relation be total on the domain of *S*; that is, *P* must produce some result whenever *S* does. The combination of partial correctness and termination is usually called 'total

If we are only interested in external behavior of a procedure that is part of a stateless module, the only state we care about is the arguments and results of the procedure. In this case, a transition is completely described by a single entry in a trace, such as `(Read(a1),d)`.

Now we are ready to consider modules with state. Our idea is to generalize what we did with pairs of states described by single trace entries to sequences of states described by longer traces. Suppose that *T* and *S* are any modules that have the same external interface (set of procedures that are exported and hence may be invoked externally). In this discussion, we will often refer to *S* as the *specification* module and *T* as the *implementation*. Then we say that *T implements S* if every trace of *T* is also a trace of *S*. That is, the set of traces generated by *T* is a subset of the set of traces generated by *S*.

This says that any external behavior of the implementation *T* must also be an external behavior of the spec *S*. Another way of looking at this is that we shouldn't be able to tell by looking at the implementation that we aren't looking at the spec, so we have to be able to explain every behavior of *T* as a possible behavior of *S*.

The reverse, however, is not true. We do not insist that the implementation must exhibit every behavior allowed by the spec. In the case of the simple memory the spec is completely deterministic, so the implementations cannot take advantage of this freedom. In general, however, the spec may allow lots of behaviors and the implementation choose just one. The spec for sorting, for instance, allows any sorted sequence as the result of `Sort`; there may be many such sequences if the ordering relation is not total. The implementation will usually be deterministic and return exactly one of them, so it doesn't exhibit all the behavior allowed by the spec.

Safety and liveness

Just as with procedures, this subset requirement is not strong enough to satisfy our intuitive notion of implementation. In particular, it allows the set of traces generated by *T* to be empty; in other word, the implementation might do nothing at all, or it might do some things and then stop. As we saw, the analog of this for a simple sequential procedure is non-termination. Usually we want to say that the implementation of a procedure should terminate, and similarly we want to say that the implementation of a module should keep doing things. More generally, when we

have concurrency we usually want the implementation to be *fair*, that is, to eventually service all its clients, and more generally to eventually make any transition that continues to be enabled.

It turns out that any external behavior (that is, any set of traces) can be described as the intersection of two sets of traces, one defined by a *safety* property and the other defined by a *liveness* property.[1] A safety property says that in the trace nothing bad ever happens, or more precisely, that no bad transition occurs in the trace. It is analogous to partial correctness for a stateless procedure; a state machine that never makes a bad transition can define any safety property. If a trace doesn't satisfy a safety property, you can always find this out by looking at a finite prefix of the trace, in particular, at a prefix that includes the first bad transition.

A liveness property says that in the trace something good *eventually* happens. It is analogous to termination for a stateless procedure. You can never tell that a trace doesn't have a liveness property by looking at a finite prefix, since the good thing might happen later. A liveness property cannot be defined by a state machine. It is usual to express liveness properties in terms of *fairness*, that is, in terms of a requirement that if some transition stays enabled continuously it eventually occurs (weak fairness), or that if some transition stays enabled intermittently it eventually occurs (strong fairness).

With a few exceptions, we don't deal with liveness in this course. There are two reasons for this. First, it is usually not what you want. Instead, you want a result within some time bound, which is a safety property, or you want a result with some probability, which is altogether outside the framework we have set up. Second, liveness proofs are usually hard.

Abstraction functions and simulation

The definition of 'implements' as inclusion of external behavior is a sound formalization of our intuition. It is difficult to work with directly, however, since it requires reasoning about infinite sets of infinite sequences of actions. We would like to have a way of proving that *T* implements *S* that allows us to deal with one of *T*'s actions at a time. Our method is based on *abstraction functions*.

An abstraction function maps each state of the implementation *T* to a state of the specification *S*. For example, each state of the `WBCache` module gets mapped to a state of the `Memory` module. The abstraction function explains how to interpret each state of the implementation as a state of the specification. For example, Figure 3 depicts part of the abstraction function from `WBCache` to `Memory`. Here is its definition in Spec, copied from handout 5.

```
FUNC AF() -> M = RET (\ a | c!a => c(a) [*] m(a) )
```

[1] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing* **2**, 3 (1987), pp 117-126.

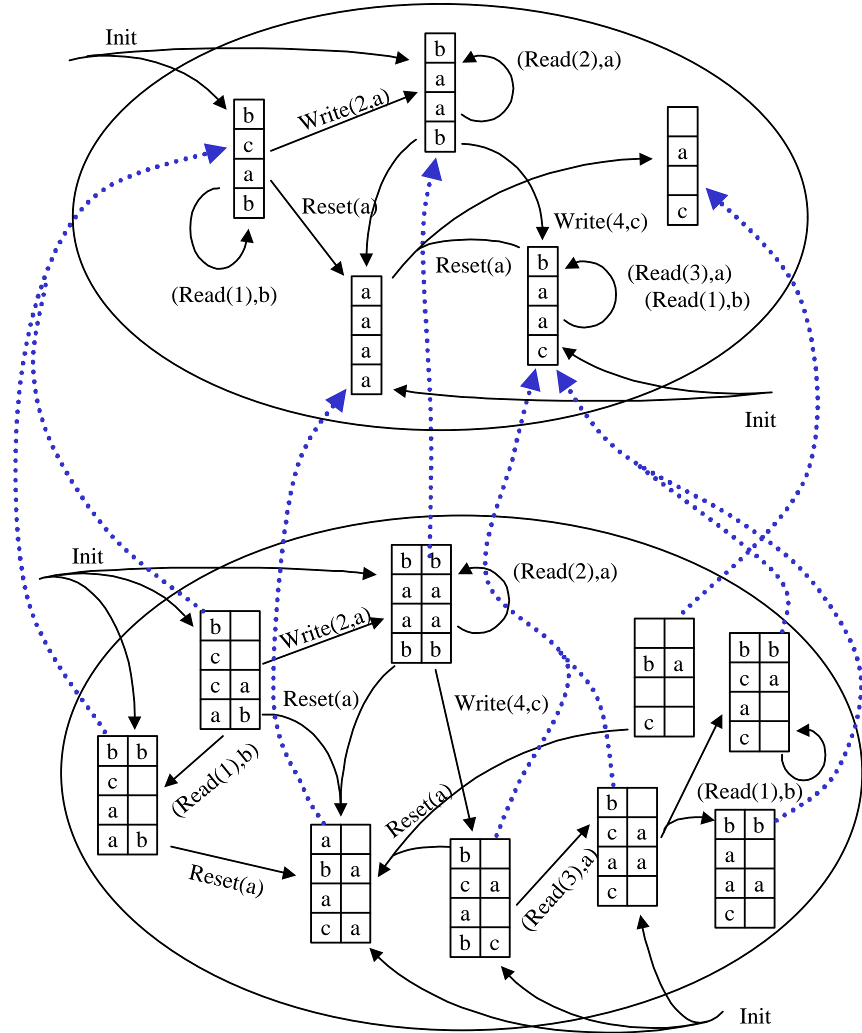

Figure 3: Abstraction function for `WBCache`

You might think that an abstraction function should map the other way, from states of the specification to states of the implementation, explaining how to represent each state of the specification. This doesn't work, however, because there is usually more than one way of representing each state of the specification. For example, for the `WBCache` implementation of `Memory`, if an address is in the cache, then the value stored for that address in memory does not matter. There are also choices about which addresses appear in the cache. Thus, many states of the implementation can represent the same state of the specification. In other words, the abstraction function is many-to-one.

An abstraction function F is required to satisfy the following two conditions.

1. If t is any initial state of T, then $F(t)$ is an initial state of S.
2. If t is a reachable state of T and (t, π, t') is a step of T, then there is a step of S from $F(t)$ to $F(t')$, having the same trace.

Condition 2 says that T *simulates* S; every step of T faithfully copies a step of S. It is stated in a particularly simple way, forcing the given step of T to simulate a single step of S. That is enough for the special case we are considering right now. Later, when we consider concurrent invocations for modules, we will generalize condition 2 to allow any number of steps of S rather than just a single step.

The diagram in Figure 4 represents condition 2. The dashed arrows represent the abstraction function F, 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 arrows, you will end up in the same state regardless of which way you go.

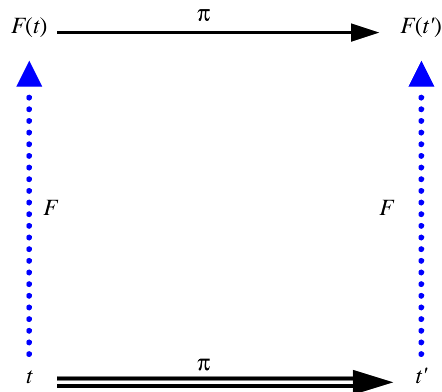

Figure 4: Commutative diagram for correctness

An abstraction function is important because it can be used to show that one module implements another:

Theorem 1: If there is an abstraction function from T to S, then T implements S, i.e., every trace of T is a trace of S.

Note that this theorem applies to both finite and infinite traces.

Proof: (Sketch) Let β be any trace of T, and let α be any execution of T that generates trace β. Use Conditions 1 and 2 above to construct an execution α' of S with the same trace. That is, if t is the initial state of α, then let $F(t)$ be the initial state of α'. For each step of α in turn, use Condition 2 to add a corresponding step to α'.

More formally, this proof is an induction on the length of the execution. Condition 1 gives the basis: any initial state of T maps to an initial state of S. Condition 2 gives the inductive step: if we have an execution of T of length n that simulates an execution of S, any next step by T simulates a next step by S, so any execution of T of length $n+1$ simulates an execution of S.

We would like to have an inverse of Theorem 1: if every trace of T is a trace of S, then there is an abstraction function that shows it. This is not true for the simple abstraction functions and simulations we have defined here. Later on, in handout 8, we will generalize them to a simulation method that is strong enough to prove that T implements S whenever that is true.

Invariants

An *invariant* of a module is any property that is true of all *reachable* states of the module, i.e., all states that can be reached in executions of the module (starting from initial states). Invariants are important because condition 2 for an abstraction function requires us to show that the implementation simulates the spec from every reachable state, and the invariants characterize the reachable states. It usually isn't true that the implementation simulates the spec from every state.

Here are examples of invariants for the `HashMemory` and `MajorityRegister` modules, written in Spec and copied from handout 5.

```
FUNC HashMemory.Inv(nb: Int, m: HashT, default: D) -> Bool = RET
     (  nb > 0
     /\ m.size = nb
     /\ (ALL a | a.hf IN m.dom)
     /\ (ALL i :IN m.dom, p :IN m(i).rng | p.a.hf = i)
     /\ (ALL a | { j :IN m(a.hf) | m(a.hf)(j).a = a }.size <= 1) )

FUNC MajorityRegister.Inv(m: M) -> Bool = RET
        (ALL p :IN m.rng, p' :IN m.rng | p.seqno = p'.seqno ==> p.d = p'.d)
     /\ (EXISTS maj | (ALL i :IN maj, p :IN m.rng | m(i).seqno >= p.seqno)))
```

For example, for the `HashMemory` module, the invariant says (among other things) that a pair containing address `a` appears only in the appropriate bucket `a.hf`, and that at most one pair for an address appears in the bucket for that address.

The usual way to prove that a property P is an invariant is by induction on the length of finite executions leading to the states in question. That is, we must show the following:

(Basis, length = 0) P is true in every initial state.

(Inductive step) If (t, π, t') is a transition and P is true in t, then P is also true in t'.

Not all invariants are proved directly by induction, however. It is often better to prove invariants in groups, starting with the simplest invariants. Then the proofs of the invariants in the later groups can assume the invariants in the earlier groups.

Example: We sketch a proof that the property `MajorityRegister.Inv` is in fact an invariant.

Basis: In any initial state, a single (arbitrarily chosen) default value d appears in all the copies, along with the `seqno` 0. This immediately implies both parts of the invariant.

Inductive step: Suppose that (t, π, t') is a transition and Inv is true in t. We consider cases based on π. If π is an invocation or response, or the body of a `Read` procedure, then the step does not affect the truth of Inv. So it remains to consider the case where π is a `Write`, say `Write(d)`.

In this case, the second part of the invariant for t (i.e., the fact that the highest `seqno` appears in more than half the copies), together with the fact that the `Write` reads a majority of the copies, imply that the `Write` obtains the highest `seqno`, say i. Then the new `seqno` that the `Write` chooses must be the new highest `seqno`. Since the `Write` writes i+1 to a majority of the copies, it ensures the second part of the invariant. Also, since it associates the same `d` with the `seqno` i+1 everywhere it writes, it preserves the first part of the invariant.

Proofs using abstraction functions

Example: We sketch a proof that the function `WBCache.AF` given above is an abstraction function from `WBCache` to `Memory`. In this proof, we get by without any invariants.

For Condition 1, suppose that t is any initial state of `WBCache`. Then `AF`(t) is some (memory) state of `Memory`. But all memory states are allowable in initial states of `Memory`. Thus, `AF`(t) is an initial state of `Memory`, as needed. For Condition 2, suppose that t and `AF`(t) are states of `WBCache` and `Memory`, respectively, and suppose that (t, π, t') is a step of `WBCache`. We consider cases, based on π.

For example, suppose π is `Read(a)`. Then the step of `WBCache` may change the cache and memory by writing a value back to memory. However, these changes don't change the corresponding abstract memory. Therefore, the memory correspondence given by `AF` holds after the step. It remains to show that both `Reads` give the same result. This follows because:

The `Read(a)` in `WBCache` returns the value t.`c(a)` if it is defined, otherwise t.`m(a)`.

The `Read(a)` in `Memory` returns the value of `AF`(t).`m(a)`.

The value of `AF`(t).`m(a)` is equal to t.`c(a)` if it is defined, otherwise t.`m(a)`. This is by the definition of `AF`.

For another example, suppose π is `Write(a,d)`. Then the step of `WBCache` writes value `d` to location `a` in the cache. It may also write some other value back to memory. Since writing a value back does not change the corresponding abstract state, the only change to the abstract state is that the value in location `a` is changed to `d`. On the other hand, the effect of `Write(a,d)` in `Memory` is to change the value in location `a` to `d`. It follows that the memory correspondence, given by `AF`, holds after the step.

We leave the other cases, for the other types of operations, to the reader. It follows that `AF` is an abstraction function from `WBCache` to `Memory`. Then Theorem 1 implies that `WBCache` implements `Memory`, in the sense of trace set inclusion.

Example: Here is a similar analysis for `MajorityRegister`, using `MajorityRegister.AF` as the abstraction function.

```
FUNC AF() -> D = RET m.rng.fmax((\ p1, p2 | p1.seqno <= p2.seqno)).d
```

This time we depend on the invariant `MajorityRegister.Inv`. Suppose π is `Read(a)`. No state changes occur in either module, so the only thing to show is that the return values are the same in both modules. In `MajorityRegister`, the `Read` collects a majority of values and returns a value associated with the highest `seqno` from among that majority. By the invariant that says that the highest `seqno` appears in a majority of the copies, it must be that the `Read` in fact obtains the highest `seqno` that is present in the system. That is, the `Read` in `MajorityRegister` returns a value associated with the highest `seqno` that appears in state t.

On the other hand, the `Read` in `Register` just returns the value of the single variable `x` in state s. Since `AF`(t) = s, it must be that s.`x` is a value associated with the highest `seqno` in t. But the uniqueness invariant says that there is only one such value, so this is the same as the value returned by the `Read` in `MajorityRegister`.

Now suppose π is `Write(d)`. Then the key thing to show is that `AF`(t') = s'. The majority invariant implies that the `Write` in `MajorityRegister` sees the highest `seqno` `i` and thus `i+1` is the new highest `seqno`. It writes `(i+1, d)` to a majority of the copies. On the other hand, the `Write` in `Register` just sets `x` to `d`. But clearly `d` is a value associated with the largest `seqno` after the step, so `AF`(t') = s' as required.

It follows that `AF` is an abstraction function from `MajorityRegister` to `Register`. Then Theorem 1 implies that `MajorityRegister` implements `Register`.