Lecture Notes: Abstract Data Types and Type States

Construction and Verification of Software 2020-21
João Costa Seco, Bernardo Toninho, Luís Caires

Version 1.0: 13 April 2021
Version 2.0: 27 April 2021 - included second handout

1 Introduction

State changes is one of the most important aspects to be captured in a software system and is usually part of software engineering basic diagrams. For instance, UML state diagrams are used to capture the life cycle of entities by identifying states and state transitions by means of system operations. Also, identifying the conditions upon which these transitions can occur. The diagram in Figure 1 depicts the states of an ATM machine, groups of states, and operations that make the transition of a machine from one state to another.

These diagrams are usually documentation to inform the development process, and can, in some occasions, generate code that controls or implements part of the behaviour. In this lecture we will encode state diagrams in our setting so that the code of an ADT can be verified against such states and state changes.

2 Abstract and Representation Invariants

So far, we have designed ADTs to explicitly have as pre- and post-conditions, an Invariant. A representation invariant restricting the values of the representation type that form a valid ADT value, and an abstract invariant that maps the valid states of an ADT to a observable layer. In the previous lectures, we've used ghost variables to represent the state of an ADT (a native dafny set), which is very expressive.

```
method add(v: int)
    requires AbsInv() ∧ size() < maxSize()
    ensures AbsInv() ∧ s = old(s) + {v}
```

This post-condition allows a client module to determine the exact elements of the ADT, without having to know the internal structure of the ADT. Notice that the pre-condition uses functions `size` and `maxSize` to determine the legal states where the function can be called.

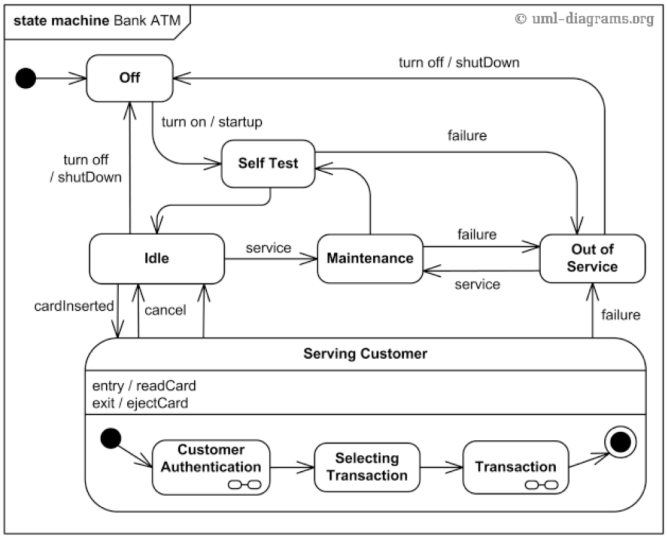

Figure 1: ATM state diagram (from `www.uml-diagrams.org`).

3 Refining ADT States

The condition to apply method `add` above is to have enough space to store another element. Which can be easily abstracted by a refined state called `NotFull`, not revealling that there is a value for a `maxSize`.

Consider now a general resource, implemented in the following `Dafny` class with the following interface

```
class Resource {

    function OpenState(): bool

    function ClosedState(): bool

    constructor()
    ensures  ClosedState();

    method Use(K: int)
    requires OpenState()
    ensures  OpenState()

    method Open()
    requires ClosedState()
    ensures  OpenState()
```

```
    method Open()
    requires OpenState()
    ensures  ClosedState()
}
```

Notice the two predicates (boolean functions) that denote two possible states for the resource. We call this states, type states. And that all methods and constructor have as pre-conditions and post-conditions the allowed starting and end states.

So, it is possible to verify that the client code

```
method UsingTheResourceOk()
{
    var r: Resource := new Resource();
    r.Open();
    r.Use(2);
    r.Use(9);
    r.Close();
}
```

is legal according to the laws of the ADT, and that the client code below is not

```
1  method UsingTheResourceNotOk()
2  {
3      var r: Resource := new Resource();
4      r.Close();
5      r.Open();
6      r.Use(2);
7      r.Use(9);
8      r.Close();
9      r.Use(2);
10 }
```

Notice that the call in line 4 is performed in a state where the resource is closed, thus not satisfying its pre-condition. Also, the call in line 9 is done after the resource has been closed which also does not satisfy its pre-conditions.

The definition of the ADT's interface above is therefore compatible with the representation in Figure 2

To ensure the correct functioning of the ADT, the functions defining the type states must imply the representation invariant of the ADT. The set of type states thus forms the abstract state of the ADT, which can also include ghost values.

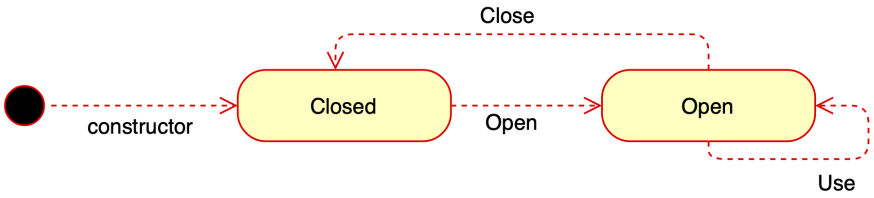

Figure 2: Resource state diagram

4 Exercises

4.1 Second Handout

This exercise is your second handout. It is due on Friday, May 7th, at 23h59m. The exercise consists of implementing, specifying and verifying a FIFO queue ADT, realized by two LIFO stacks, achieving $O(1)$ amortized worst case complexity for both enqueue and dequeue operations.

The implementation consists of having an input stack, where method `enqueue` inserts elements into; and an output stack, where method `dequeue` takes elements from. The communication between the two stacks is performed when the dequeue operation encounters the output stack empty. At this time, all elements from the input stack are popped and pushed to the output stack, resulting in the output stack containing the elements of the input stack in reverse order (and so the top element of the output stack is the element that is to be dequeued). The stacks should be implemented with an array and an integer denoting the number of elements stored in each of them (i.e. the integer "points to" the array element where the next push takes place).

Your specification should use an abstract representation with a ghost variable containing a sequence of integers to prove that the ADT behaves indeed like a FIFO queue. The abstract state should be used to specify correctly the `enqueue` and `dequeue` operations.

Use typestates to discipline the use of the queue based on the abstract state and refining the representation invariant. You should also present a client method that conveniently instantiates and uses a queue. Statically verify as many calls as you can using type states.

4.2 Interface and hints

Consider the following interface for your code. Some hints are included in the code below (you will need to provide bodies for the missing predicates and methods, as well as any missing pre- and post-conditions). You must use the provided soundness mapping that relates the representation type and the abstract state, and also the provided definition of the reverse function (this will ease the verification effort).

```
class Queue {
```

```
// Representation types
var input: array<int>;
var in_n: int;
var output: array<int>;
var out_n: int;
//The top of the stack is maintained on the higher indices of the
//array.

// Abstract state, a sequence in Dafny
// [0] <- [1,2,3,4,5] <- [6]
// Dequeue from the left, enqueue on the right.
ghost var q : seq<int>;

function size() : int

predicate RepInv()
// Hint: remember to control the aliasing between the two
//  arrays. This is crucial for the verification to work.

function reverse(s: seq<int>) : seq<int>
    decreases s
{if s = [] then [] else s[ |s| - 1.. |s| ] + reverse(s[..|s| - 1 ])}
// Please use this definition of the reverse function.

predicate Sound()
    requires RepInv()
    reads this, output, input
{ q = reverse(output[..out_n]) + input[..in_n] }
// The correspondence between the arrays and the queue
// must be maintained at all times.

predicate AbsInv()
    reads this, output, input
{ RepInv() ∧ Sound() }

predicate NotEmpty()

predicate NotFull()

constructor(n: int)
    requires n > 0;
    ensures AbsInv();
```

```
    method enqueue(x: int)
        requires NotFull()
        ensures AbsInv()
        ensures q = old(q) + [x]

    method dequeue() returns (x: int)
        requires NotEmpty()
        ensures AbsInv()
        ensures [x] + q = old(q)
    // Method dequeue calls method flush whenever the output stack
    // is empty and there are elements in the input stack.

    method flush()
        requires AbsInv() ∧ out_n = 0
        ensures AbsInv()
        ensures q = reverse(output[..out_n])
    // It copies all the elements from one stack to the other
    // maintaining the order in the queue. Try and relate
    // the steps in the schema below with the representation invariant
    // to determine the appropriate loop invariant.

        // out: []->  in: [1,2,3,4,5]<-
        // out: [5]     in: [1,2,3,4]
        // out: [5,4]     in: [1,2,3]
        // out: [5,4,3]     in: [1,2]
        // out: [5,4,3,2]     in: [1]
        // out: [5,4,3,2,1]->  in: []

}
```

In many situations in this proof, it helps to include asserts that identify the separation of sequences into parts. For instance, it will often be helpful to state that the output stack, when seen as a sequence, consists of the concatenation of sequences `output[..out_n-1]` and `output[out_n-1..out_n]`:

```
assert output[..out_n] == output[..out_n-1]+output[out_n-1..out_n];
```

This kind of "obvious" decomposition allows Dafny to more adequately automate reasoning when adding or removing an element to the `output` array. Similar techniques apply to the `input` array.

A Queue

In this section you can find the source of the example shown in the live lecture.

```
class Queue {

    var a: array<int>;
    var front: int;
    var rear: int;

    var numberOfElements: int;

    function RepInv(): bool
        reads this
    {
        0 < a.Length ∧
        0 ≤ front < a.Length ∧
        0 ≤ rear < a.Length ∧
        if front = rear then
            numberOfElements = 0 ∨
            numberOfElements = a.Length
        else
            numberOfElements =
                if front > rear then front - rear
                                else front - rear + a.Length
    }

    function NotEmpty():bool
        reads this
    {
        RepInv() ∧ numberOfElements > 0
    }

    function NotFull():bool
        reads this
    {
        RepInv() ∧ numberOfElements < a.Length
    }

    function method Full():bool
        reads this
    {
        numberOfElements = a.Length
    }
```

```
    function method Empty(): bool
        reads this
    {
        numberOfElements = 0
    }

    constructor(N: int)
        requires 0 < N
        ensures fresh(a)
        ensures NotFull()
    {
        a := new int[N];
        front := 0;
        rear := 0;
        numberOfElements := 0;
    }

    method Enqueue(V: int)
        modifies this`front, this`numberOfElements, a
        requires NotFull()
        ensures NotEmpty()
    {
        a[front] := V;
        front := (front + 1)%a.Length;
        numberOfElements := numberOfElements + 1;
    }

    method Dequeue() returns (V: int)
        modifies this`rear, this`numberOfElements, a
        requires NotEmpty()
        ensures NotFull()
    {
        V := a[rear];
        rear := (rear + 1)%a.Length;
        numberOfElements := numberOfElements - 1;
    }
}

method Main()
{
    var q: Queue := new Queue(4);
    var r: int;
```

```
    q.Enqueue(1);
    r := q.Dequeue();
    q.Enqueue(2);
    r := q.Dequeue();
    q.Enqueue(3);
    if (¬q.Full())
    { q.Enqueue(4); }
    r := q.Dequeue();
    if (¬q.Empty())
    { r := q.Dequeue(); }
    // q.Enqueue(5);
    // q.Enqueue(5);
    // q.Enqueue(5);
    // q.Enqueue(5);
    // q.Enqueue(5);
}
```