

# Chapter 6

## The Universality of Consensus

### 6.1 Introduction

In the previous chapter, we considered a simple technique for proving statements of the form “there is no wait-free implementation of  $X$  by  $Y$ .” We considered object classes with deterministic sequential specifications<sup>1</sup>. We derived a hierarchy among them in which no combination of objects from one level can implement an object that is at a higher level (see Figure 6.1). Recall that each object class has an associated *consensus number*, which is the maximum number of threads for which the object can solve the consensus problem. In a system of  $n$  or more concurrent threads, it is impossible to construct a wait-free implementation of an object with consensus number  $n$  from objects with a lower consensus number.

These impossibility results do not by any means imply that wait-free syn-

---

<sup>1</sup>We avoid the case of non-deterministic objects since it is significantly more complex and riddled with anomalies.

Consensus Number	Object
1	read/write registers
2	test&set, swap, fetch&add, queue, stack
⋮	⋮
$2n - 2$	$n$ -register assignment
⋮	⋮
$\infty$	memory-to-memory move and swap, augmented queue, compare&swap, fetch&cons, sticky byte

Figure 6.1: Impossibility and Universality Hierarchy

chronization is impossible or infeasible. In this chapter, we show that there exist *universal* classes of objects from which one can construct a wait-free linearizable implementation of any object. We give a simple test for universality, showing that a class is universal in a system of  $n$  threads if and only if it has a consensus number greater than or equal to  $n$ . In Figure 6.1, each class at level  $n$  is universal for a system of  $n$  threads. A machine architecture or programming language is computationally powerful enough to support arbitrary wait-free (or lock-free) synchronization if and only if it provides objects of a universal class as a primitive.

## 6.2 Universality Results

A class  $C$  is *universal* if one can construct a wait-free implementation of any object from some number of objects of  $C$  and any amount of read/write registers. We allow an implementation to use multiple objects of class  $C$  because we are ultimately interested in understanding the synchronization power of machine instructions, and most machines allow their instructions to be applied to multiple memory locations. We also allow an implementation to use read/write registers because it is convenient for bookkeeping, and memory is usually in plentiful supply on modern architectures.

We will see that any class with consensus number  $n$  is universal in a system of  $n$  (or fewer) threads. The basic idea is the following: we represent the object to be implemented as a linked list, where the sequence of cells represents the sequence of method calls applied to the object (and hence the object's sequence of states). A thread executes a method call by threading a new cell on to the end of the list. When the cell becomes sufficiently old, it can be reclaimed and reused.

This chapter is not intended to provide practical techniques for constructing wait-free object implementations. Instead, it is intended to tell us “where to look” for such techniques. If we understand *why* consensus is powerful enough to implement any kind of object, then we are better prepared to undertake the engineering effort needed to make such constructions efficient.

## 6.3 Sequential Objects

We start out with a sequential, unsynchronized implementation of the object for which we want to construct a wait-free linearizable implementation. It is convenient to cast this sequential implementation into a kind of “generic form”, illustrated in Figure 6.2. An unusual aspect of this generic form is that sequential objects are *immutable*: a method call does not modify a sequential object, instead it creates a new sequential object whose state reflects the effects of that

---

<sup>1</sup>This chapter is part of the Manuscript MULTIPROCESSOR SYNCHRONIZATION by Maurice Herlihy and Nir Shavit copyright © 2003, all rights reserved.

```

public class SeqObject {
    public class Invoc {
        public String method;
        public Object[] arguments;
    }
    public class Response { // struct-like class
        public SeqObject object; // new object state
        public Object value; // return value
    }
    public Response apply(Invoc invoc) { // returns new state and result
        return null;
    }
}

```

Figure 6.2: generic sequential implementation

method call. For example, a `deq` method call would return both the item dequeued from the queue, and a new version of the queue where that item had been removed.

The `SeqObject` class exports two inner classes, `Invoc` and `Response`. The `Invoc` class includes the name of the method being invoked (expressed as a string) and its arguments (an array of `Object`). The `Response` class includes the return value from the method invocation, as an `Object`, which could represent an exception, and a new `SeqObject` reflecting the effect of the method call. The `SeqObject` class provides one method: `apply`, which takes an `Invoc` object as its only argument, and returns a `Response` object.

## 6.4 Cells

The object itself is represented by a doubly-linked list of `Cell` objects having the following fields:

- `seq` is the cell's sequence number in the list. This field is zero if the cell is initialized but not yet threaded onto the list, and otherwise it is positive. Sequence numbers for successive cells in the list increase by one.
- `invoc` is the `Invoc` object for the method call.
- `response` is the `Response` object for the method call.
- `nextState` is the method call's resulting state.
- `nextCell` is a consensus object used to decide which cell is placed next in the list.

```
public class Cell {

    public int seq;           // sequence number
    public Invoc   invoc;    // method name and args
    public Response response; // object state and response
    public SeqObject nextState; // next object state
    public Consensus nextCell; // next Cell in list

    public Cell(Invoc invoc) {
        this.invoc      = invoc;
        this.response   = null;
        this.nextState  = null;
        this.nextCell   = new Consensus();
        this.seq        = 0;
    }

    public Cell max(Cell aCell) {
        if (this.seq > aCell.seq)
            return this;
        else
            return aCell;
    }
}
```

Figure 6.3: cell

The constructor for a `Cell` object takes an `Invoc` object, initializes `nextCell` field to a new consensus object, and sets the `seq` field to zero. The `Cell` class also provides a `max` method that compares two cells and returns the one with the higher sequence number. Initially, the object is represented by a unique *anchor* cell with sequence number 1, holding a constructor call and an initial state.

## 6.5 The Algorithm

The full protocol appears in Figure 6.4. The threads share the following data structures.

- `announce` is an  $n$ -element array, where `announce`[ $i$ ] is the cell thread  $i$  is currently trying to append to the list. Initially all elements point to the anchor cell.
- `head` is an  $n$ -element array, where `head`[ $i$ ] is the last cell in the list that thread  $i$  has observed. Initially all elements point to the anchor cell.

It is convenient to define some notation. Let  $\max(\text{head})$  be the cell with the largest sequence number in the `head` array, and let “ $c \in \text{head}$ ” denote the assertion that cell  $c$  has been assigned to `head`[ $i$ ], for some  $i$ . An *announcement* by  $A$  occurs when  $A$  stores a new cell in the `announce` array.

An *auxiliary* variable is one that does not appear explicitly in the code, but that helps us reason about the behavior of the protocol. We use the following auxiliary variables:

- $\text{concur}(A)$  is the set of cells that have been stored in the `head` array since  $A$ ’s last announcement.
- $\text{start}(A)$  is the the value of  $\max(\text{head})$  at  $A$ ’s last announcement.

Notice that:

$$|\text{concur}(P)| + \text{start}(P) = \max(\text{head}) \quad (6.1)$$

Auxiliary variables do not affect the protocol’s control flow; they are present only to facilitate proofs.

Informally, the protocol works as follows.  $A$  allocates and initializes a cell to represent the method call (line 11). It stores (a pointer to) the cell in `announce`[ $A$ ] (line 11), ensuring that if  $A$  itself does not succeed in appending its cell onto the list, some other thread will append that cell on  $A$ ’s behalf. To locate a cell near the end of the list,  $A$  reads the entries of the `head` array, setting `head`[ $A$ ] to the cell with the maximal sequence number (line 13).  $A$  then enters the main loop of the protocol (line 14), which it executes until its own cell has been threaded onto the list (detected when its sequence number becomes non-zero).  $A$  first chooses a thread to “help” (line 18), and checks whether that thread has an unthreaded cell (line 19). If so, then  $P$  will try to thread it, otherwise it tries to thread its own cell. (If this helping step were omitted,

```

public class Universal {

    private Cell[] announce;
    private Cell[] head;

    4 public Response apply(Invoc invoc) {
        int i = Thread.myIndex();
        Cell prefer;

11  this.announce[i] = new Cell(invoc);
12  for (int j = 0; j < n; j++)
13      this.head[i] = this.head[i].max(this.head[j]);
14  while(this.announce[i].seq == 0) {
        // pick a cell near the end
        Cell before = this.head[i];
        // pick someone to help
18      Cell help = this.announce[(before.seq + 1) % n];
19      if (help.seq == 0)
            prefer = help;
        else
            prefer = this.announce[i];
        // try to append cell to list
24      before.nextCell.propose(prefer);
25      Cell after =(Cell)before.nextCell.decide();
        // fill in remaining fields of after cell
27      SeqObject oldObject = before.response.object;
        Response response = oldObject.apply(after.invoc);
        after.nextState = response.object;
30      after.response = response.value;
        this.head[i] = after;
        after.seq = before.seq + 1;
    }
    this.head[i] = this.announce[i];
    return this.announce[i].response;
}
}

```

Figure 6.4: universal protocol

then an individual thread could be overtaken an arbitrary number of times.) The `prefer` variable names the cell that  $A$  will try to append.  $A$  then proposes `prefer` to the consensus object `before.nextCell`, and then calls `decide` to learn the next cell (lines 24 and 25). The `nextCell` field must be a consensus object to ensure that there is no ambiguity about the order of the cells in the list. Independently of the next cell's identity,  $A$  proceeds to fill in the remaining fields (lines 27-30). The response is stored in the `response` field for later reference. Finally, the cell's sequence number is initialized. Notice that multiple threads may write to the cell's `response` and `seq` fields, but they always write the same value, so they do not need to be consensus objects.

For brevity, we say that a thread *appends* a cell if the `decide` method returns that thread's argument to `propose` in lines 24 and 25. A thread *announces* a cell at line 11 when it stores the cell in the `announce` array.

**Lemma 6.5.1** *The following claim is always true:*

$$|\text{concur}(A)| > n \Rightarrow \text{announce}(A) \in \text{head}$$

*Proof:* If  $|\text{concur}(A)| > n$ , then  $\text{concur}(A)$  includes successive cells  $b$  and  $c$  with respective sequence numbers equal to  $A - 1 \bmod n$  and  $A \bmod n$ , appended by threads  $B$  and  $C$ . Because  $b$  is in  $\text{concur}(A)$ ,  $B$  appends  $b$  after  $A$ 's announcement. Because  $C$  cannot modify an unappended cell,  $C$  reads  $\text{announce}[A]$  after  $B$  appends  $b$ . It follows that  $C$  reads  $\text{announce}[A]$  after  $A$ 's announcement, and therefore either  $\text{announce}[A]$  is already appended, or  $c$  is  $\text{announce}[A]$ . ■

Lemma 6.5.1 places a bound on the number of cells that can be appended while a method call is in progress. We now give a sequence of lemmas showing that when  $P$  finishes scanning the `head` array, either  $\text{announce}[A]$  is appended, or  $\text{head}[A]$  lies within  $n + 1$  cells of the end of the list.

**Lemma 6.5.2** *The following property always holds:*

$$\max(\text{head}) \geq \text{start}(A).$$

*Proof:* The sequence number for each  $\text{head}[B]$  is non-decreasing. ■

**Lemma 6.5.3** *The following is a loop invariant for line 14 (it holds during each iteration of the loop).*

$$\max(\text{head}[A], \text{head}[i], \dots, \text{head}[n]) \geq \text{start}(A).$$

where  $i$  is the loop index.

*Proof:* When  $i$  is 0, the assertion is implied by Lemma 6.5.2. The truth of the assertion is preserved at each iteration, when  $\text{head}[A]$  is replaced by  $\max(\text{head}[A], \text{head}[i])$ . ■

**Lemma 6.5.4** *The following assertion holds just before line 14:*

$$\text{head}[A].\text{seq} \geq \text{start}(A).$$

*Proof:* After the loop at line 12,  $\max(\text{head}[P], \text{head}[Q], \dots, \text{head}[n])$  is just  $\text{head}[A].\text{seq}$ , and the result follows from Lemma 6.5.3. ■

**Lemma 6.5.5** *The following property always holds:*

$$|\text{concur}(A)| \geq \text{head}[A].\text{seq} - \text{start}(A) \geq 0$$

*Proof:* The lower bound follows from Lemma 6.5.4, and the upper bound follows from Equation 6.1. ■

**Theorem 6.5.6** *The protocol in Figure 6.4 is correct and bounded wait-free.*

*Proof:* Linearizability is immediate, since the order in which cells are appended is clearly compatible with the natural partial order of the corresponding method calls.

The protocol is bounded wait-free because  $A$  can execute the main loop no more than  $n + 1$  times. At each iteration,  $\text{head}[A].\text{seq}$  increases by one. After  $n + 1$  iterations, Lemma 6.5.5 implies that

$$|\text{concur}(A)| \geq \text{head}[A].\text{seq} - \text{start}(A) \geq n.$$

Lemma 6.5.1 implies that  $\text{announce}[A]$  must have been appended. ■