

# Notes on modularity in digital systems and programs

Victor Yodaiken\*

FSResearch LLC ,  
2718 Creeks Edge Parkway  
yodaiken@fsmllabs.com  
<http://www.yodaiken.com>

**Abstract.** The structure of state machines reveals why modularity is such a key design advantage in programming and computer engineering, why it is so difficult, and how surface modularity works. I have no idea whether this is well known material, but have not seen it anywhere and think it's interesting so, here it is.

**Key words:** modularity, transducer, Moore machine, primitive recursion, composition, parallel

## 1 Introduction

Modularity is a desirable but elusive design property for large scale programs and computer systems [10, 11]. Designs that appear modular may, once put into practice, actually turn out to be example of “false modularity” because of interdependencies between components. For example, the apparently modular architecture of micro-kernel operating systems [9] has run into practical difficulties [6, 3]. Examining how state machines can be constructed by connecting simpler state machines together, provides some insight into why real modularity is so powerful and how to avoid false modularity. In this paper, I’m going to sketch out how to look at modularity from the perspective of state machines, try to derive a few “rules of thumb”, and point to some of the deep mathematical basis of this method.

The power of modularity comes from simple division. If a state machine requiring  $2^{32}$  states can be implemented by connecting 4 state machines of 256 states each, the total system may still have 4 billion or so states, but each component is fairly simple. False modularity pops up when such division just displaces the complexity to the interconnection.

---

\*

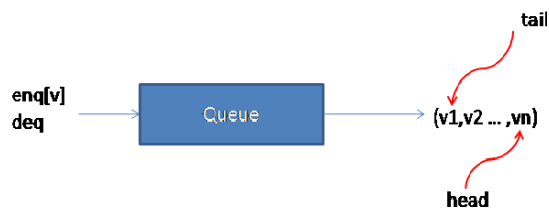
In principle, any state machine with  $n$  states can be divided into a network of machines of 2 states each with  $\lceil \log_2 n \rceil$  machines in the network. These “bit” machines encode the states of the original machine. At each step, each “bit” machine accepts an input that includes a bit from every component so that it can decode the state of the composite state system. In this case, the entire system state is broadcast to every component on each step. This is the limiting case for false modularity and gives a sense of what properties we need to look for in evaluating designs and in making sure interconnection schemes make sense.

Researchers in “abstract automata” theory in the early 1960s discovered that limiting interconnections to a pipeline or cascade had implications for how state machines could be factored into modules. In particular, they showed that loops in state machines define certain mathematical groups and where those groups are “solvable”, the machines can be chopped into pipelines, but when the groups are “simple” (primitive) the state machines cannot be factored into products. Unfortunately, the researchers in this area have not, as far as I know, produced much in the way of accessible explanations. In this work, I’m going to emphasize the engineering, not the mathematics.

## 2 Factoring out groups

To build some intuition, consider how to factor fifo-queues and then ordered-queues. These ubiquitous data structures are over-used as examples, but in this case they provide startlingly clear indication of what group theory has to do with modularity. This section stresses intuition and postpones precise definition of these state machines until section ??.

A fifo-queue state machine accepts inputs to enqueue elements and dequeue them in a first-in-first-out (fifo) order. Let  $()$  be the empty list,  $(v_1, \dots, v_n)$  be an  $n$  element list where  $v_1$  is the “tail” element and  $v_n$  is the “head”, and let  $v.L$  indicate the list obtained by pushing  $v$  onto the



**Fig. 1.** Queue

head so  $v.(v_1 \dots, v_n) = (v, v_1 \dots, v_n)$ . Let  $deq(L) = ()$  if  $L = (v)$ ,  $(v)$  if  $L = (v_1, v_2)$  and  $(v_1 \dots v_{n-1})$  if  $L = (v_1, \dots, v_n)$  for  $n > 2$ . Finally, let  $|L|$  be the length of a list with  $|()| = 0$ , and  $|(v_1 \dots v_n)| = n$ . Any feasible queue will have a maximum length. A brute force implementation of a queue has a state for each possible ordered tuple of queue contents. The initial state will be  $()$  the empty tuple. An  $enq[v]$  input in state  $L$  produces state  $v.L$  if  $|L|$  is less than the maximum queue length and leaves the state unchanged otherwise. A  $deq$  operation simply applies the function  $deq$  to the state. If the queue has maximum length  $k$  and queue elements are from a set of possible values  $V$  that has  $n$  elements, then there are  $\sum_{i=0}^k n^i$  possible states.

From engineering considerations, it is known that one can factor this machine into a memory array of  $k$  storage elements and two counters. One counter tracks how many elements are

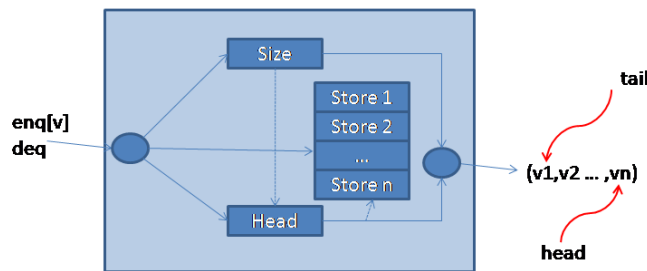


Fig. 2. Factored Queue

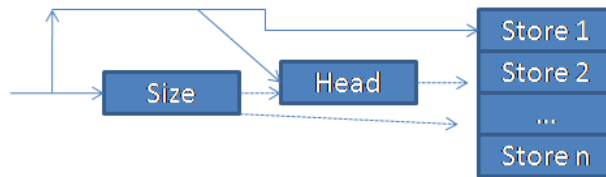
in the queue and the other follows the head of the queue around the array as elements are removed. The queue is empty if the size is 0 and full if the size is  $k$ . Number the storage elements from 0 to  $k-1$  and then the tail of the queue is storage element  $head + size \bmod k$ . Appending an element when the queue is not full increments  $size$ , leaves  $head$  unchanged, and the new value is stored in element  $tail = (head + size \bmod k)$  before  $size$  is incremented. The head of the queue when the queue is not empty is the value stored in element  $head$ . Removing an element when the queue is not empty increments  $head$  by  $1 \bmod k$  and decrements  $size$  by one (we don't need the  $-1 \bmod k$  here because we ignore enqueue requests when  $size = k$ ). We can then build the queue with  $k + 2$  state machines where each storage element has  $n$  states, the  $head$  has  $k$  states and the  $size$  has  $k + 1$  states. Total state

size of the system as a network is  $k * n + 2k + 1$  which is significantly less than  $\sum_{i=0}^k n^i$  for most interesting values of  $k$  and  $n$ .

So we took the original state machine and implemented the same functionality via  $k + 2$  simpler machines that are interconnected. Two of the state machines are counters which, as we'll see below, means that they correspond to cyclic groups. The other state machines have simple state graphs: an input  $v$  changes the state to  $v$ , no matter what the current state. These state machines are called group-free or counter free state machines. The communications between the machines are nicely limited. The counters never need to know the contents of the stores. The stores only need to know whether the input is  $enq[v]$  AND  $tail$  points to them AND  $size < k$ . The  $head$  counter needs to know the input and whether  $size < k$  or  $size > 0$ .

Note that information flows in a “cascade” from left to right in this network of machines.

h



**Fig. 3.** Queue cascade

Let's now consider an ordered or sorted queue — something that is often implemented by more complex data structures such as trees. The number of states of the brute force version of an ordered queue is smaller than the number of states of the fifo-queue because every ordered queue state is a fifo-queue state, but many fifo-queue states are not ordered. For example (100, 101, 99, 5, 80) is a disordered list that would not be a state of the sorted queue. The smaller state set, however, comes with a more complex operation: we don't just append new items to the end of the queue, they have to be inserted in place.

In this case, a simple counter is not going to suffice but we can again separate the control information from the storage of elements. The control information can be kept by a machine which maintains an ordered

list of indexes while data is stored in memory cells as in the fifo-queue. Let's assume that the *deq* operation is intended to remove the "greatest" element from the queue — making it the "least" element would be an easy change. If the storage elements are numbered  $1 \dots k$ , then we can keep all ordering information in a list

$$(i_1 \dots, i_r, 0, j_1 \dots j_d)$$

where  $r + d = k$  and the elements to the left of 0 indicate storage elements in use, each one greater than or equal to its left neighbor. The elements to the right of 0 are unused or free. When an element is inserted in the queue, we want to swap 0 with the index to its right and then keep swapping left until the list is ordered correctly again. A *deq* operation just swaps 0 with the head of the queue — the element to just to the left of 0. Each operation then *permutes* the list. Storage element  $i$  accepts a new value  $v$  when the input is *enq*[ $v$ ] and the element to the right of the 0 is  $i$ .

This decomposition is not as simple as the fifo-queue decomposition because there is some feedback. On *deq*, the control machine just swaps 0 and the element to its left, freeing the head element. But on *enq* the control machine needs to know where to put the index to the right of the 0 marker and that depends on a single bit of information from each of the allocated storage locations — the result of comparing the new value to the current value stored in the location. In many cases, 1 bit per storage cell is not much of an interconnection burden and certainly it could be implemented in ways to reduce the communication overhead even more. To me, the decision to exclusively focus on the loop-free decomposition was an error. The kind of decomposition sketched here is interesting in itself and it turns out the type of permutation group determines whether the permutation machine itself can be further factored using loop-free factorization.

### 3 To be continued

#### References

1. Michael A. Arbib. *Algebraic theory of machines, languages, and semi-groups*. Academic Press, 1968.
2. Michael A. Arbib. *Theories of Abstract Automata*. Prentice-Hall, 1969.
3. Sam Ockman Chris DiBona, editor. *Open Sources Voices from the Open Source Revolution*. O'Reilly Media, January 1999.
4. Pal Domsosi and Chrystopher L. Nehaniv. *Algebraic Theory of Automata Networks (SIAM Monographs on Discrete Mathematics and Applications, 11)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004.

5. J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, 1966.
6. Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur. Are virtual-machine monitors microkernels done right? *SIGOPS Oper. Syst. Rev.*, 40(1):95–99, 2006.
7. W.M.L. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, 1983.
8. Chrystopher L. Nehaniv John Rhodes. *Applications of Automata Theory and Algebra by John Rhodes: Book Cover Applications of Automata Theory and Algebra : Via the Mathematical Theory of Complexity to Biology, Physics, Psychology, Philosophy, and Games*. World Scientific Publishing Company, Incorporated, 2009.
9. J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, 1995.
10. D. L. Parnas. On the criteria to be used in decomposing systems into modules. pages 139–150, 1979.
11. David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. In *ICSE*, pages 408–419, 1984.