Notes on modularity in digital systems and programs

Victor Yodaiken
FSResearch LLC
@vyodaiken Twitter
`http://www.yodaiken.com`

February 9, 2016

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 false modularity works.

1 Introduction

Modularity is a desirable but elusive design property for large scale programs and computer systems [7, 1]. Designs that look modular may, once put into practice, actually turn out to be not modular because of interdependencies between components. For example, the apparently modular architecture of micro-kernel operating systems [5] runs into practical difficulties [2]. 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.

2 State machines and modularity

In principle any digital computing system can be represented as a deterministic finite state machine[1]. Factoring a state machine into components

[1]This is weirdly controversial because of some conventional wisdoms about non-determinism, communication, and parameters that are too tedious to discuss.

then corresponds to breaking the original system into modules. The power of modularity then is apparent from arithmetic. A network of n connected state machines each with J_i states can be "multiplied out" into a single state machine that has $\Pi_{i=1}^{n} J_i$ states. 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. Such systems violate the principle of "information hiding" that Parnas[7] identifies as the basis of modularity.

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. False modularity arises when we factor a system into components that have to share too much state information with each other. True modularity comes from a factorization that exhibits low bandwidth communication. That is the information communicated should be small compared to the state sizes of the components and it should perhaps be otherwise constrained.

3 Wave hands at the algebra

The relationship between automata and semigroups has been known of since the 1950s. A semigroup is just a set plus an associative binary operation. A semigroup is a monoid if it has an indentity element. It is a group if each element has an inverse. Think of each string over the event alphabet of a state machine as a map from states to states. The set of maps under composition is a semigroup. That semigroup is a monoid if we include the map on states given by empty string - which takes each state to itself. If the state set is finite, the monoid is necessarily finite as well. In the early 1960s, it was discovered that a certain product of semigroups called a cascade (or wreath) corresponded to a "loop-free" product of state machines – where information flows in a uni-directional stream between state machines[3]. The Krohn-Rhodes theorem (see [6] for a recent survey or [4] for details.) connected these semigroup products to the subgroup structure of groups. As a result, it appears that algebraic structure of semigroups and groups is related to the design of computer systems in terms of connected components.

4 Factoring

To build some intuition, consider how to factor fifo-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.

A fifo-queue state machine accepts inputs to enqueue elements and dequeue them in a first-in-first-out (fifo) order. An input $enq[v]$ should put v onto the end (tail) of the queue and an input deq should remove the first element (head) of the queue. If the queue has maximum length K and contains elements from a set of J items then there are $\Sigma_{i=0}^{K} J^i$ possible states. Even for small K and J this is a large number. Suppose we have a queue of 8 bit bytes with maximum length 20 then there are more than 10^{48} states. But from programming we know we can build this queue from an array of K memory cells each of which has J states and two additional memory cells that have K states each to act as counters. Number the array from 0 to $K-1$ and call one of the additional cells "Size" and the other "Head" and make them both initially be 0. Suppose Size holds r and Head holds m. If $r = k$ an $enq[v]$ will be ignored and if $r = 0$ a deq will be ignored. Otherwise $enq[v]$ will store v into array element $(r + m)$ mod K. In code it looks something like this.

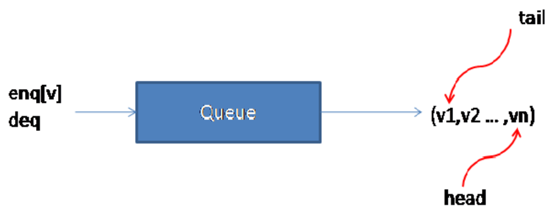

Figure 1: Queue

```
enq(v){ if(Size < K)then {
                    array[ (Size+ Head) mod K]:= v;
                    Size := Size +1 ;
   }
       }

deq() { if(Size > 0) then{
```

```
          Temp:=array[Size+Head];
          Head := Head+1 mod K
          Size := Size -1;
          }

          return Temp
      }
```

Note that there are $K * J$ states for the array elements and $2K + 1$ for the two counters $20 * 256 * 41$ is a lot less than 10^{48}. And note that there's not all that much communication needed between components. The cells in the array get values from the system input and one is connected to system output on a *deq* and *Head* and *Size* need to communicate, but only whether the queue is full or empty or neither.

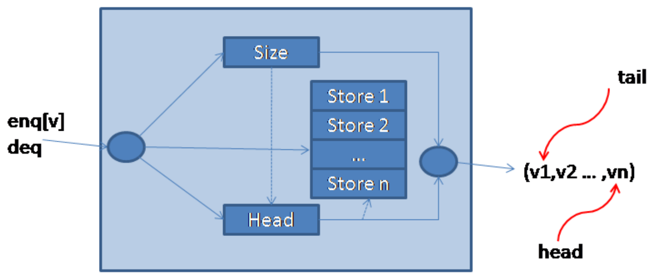

Figure 2: Factored Queue

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.

As a second example, consider a n bit binary adder. Inputs are *reset* and $\vec{b}$ for any binary n-tuple $\vec{b}$. States are all the binary n-tuples. Let $unsigned(\vec{b}) = \Sigma_{i=1}^{n} \vec{b}_i * 2^{i-1}$ where the tuple is numbered $1 \ldots n$. The input h

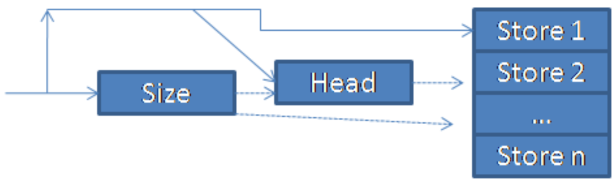

Figure 3: Queue cascade

reset moves to the state $\vec{b}$ so that $unsigned(\vec{b}) = 0$ While in state $\vec{s}$ an input $\vec{b}$ goes state $\vec{s'}$ so that $unsigned(\vec{s'}) = unsigned(\vec{s}) + unsigned(\vec{b'}) \bmod 2^n$. From engineering practice we know we can factor this 2^n state machine into n machines of 2 states each. And we know that the communication complexity builds from lower order to higher order with each machine needing the outputs of all "lower order" bits.

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. Because Krohn-Rhodes bridged automata studies to group theory, the "loop-free" uni-directional connections implicated in cascade products became the focus of algebraic automata theory, but I'm not sure that was a great idea. 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 in practice. 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.

Finally, consider a queue in which ordering is determined by input. Instead of $enq[v]$, our inputs can be $enq[p, v]$ where p is the position of the insertion point. So $enq[1, v]$ puts v at the head of the queue and $enq[4, v]$ inserts v in position 4. In this case, we have exactly the same components as the ordered queue, but we eliminate the feedback since the permutation depends entirely on the p input.

References

[1] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 195–204, Piscataway, NJ, USA, 1981. IEEE Press.

[2] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, New York, NY, USA, 2013. ACM.

[3] J. Hartmanis. Loop-free structure of sequential machines. In E.F. Moore, editor, *Sequential Machines: Selected Papers*, pages 115–156. Addison-Welsey, Reading MA, 1964.

[4] W.M.L. Holcombe. *Algebraic Automata Theory.* Cambridge University Press, 1983.

[5] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, September 1996.

[6] Oded Maler. Time for verification. chapter On the Krohn-Rhodes Cascaded Decomposition Theorem, pages 260–278. Springer-Verlag, Berlin, Heidelberg, 2010.

[7] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.