State machines for large scale computer software and systems

VICTOR YODAIKEN, Independent Researcher, USA

The behavior and architecture of large scale discrete state systems found in computer software and hardware can be specified and analyzed using a particular class of primitive recursive functions. This paper begins with an illustration of the utility of the method via a number of small examples and then via longer specification and verification of the ”Paxos” distributed consensus algorithm[Lam01]. The “sequence maps” are then shown to provide an alternative representation of deterministic state machines and products of state machines.

Distributed and composite systems, parallel and concurrent computation, and real-time behavior can all be specified naturally with these methods - which require neither extensions to the classical state machine model nor any axiomatic methods or other techniques from formal logic or other foundational methods. Compared to state diagrams or tables or the standard set-tuple-transition-maps, sequence maps are more concise and better suited to describing the behavior and compositional architecture of computer systems. Staying strictly within the boundaries of classical deterministic state machines anchors the methods to the algebraic structures of automata and makes the specifications faithful to engineering practice.

CCS Concepts: • **Theory of computation** → **Transducers**; **Formal languages and automata theory**; **Concurrency**; **Automata extensions**; **Logic**; *Modal and temporal logics*; *Logic and verification*; • **Computer systems organization** → *Dependable and fault-tolerant systems and networks*;

Additional Key Words and Phrases: automata, recursion, specification, concurrency

ACM Reference Format:

Victor Yodaiken. 2023. State machines for large scale computer software and systems. *Form. Asp. Comput.* 1, 1, Article 1 (January 2023), 28 pages. https://doi.org/10.1145/3633786

1 INTRODUCTION

This paper has three objectives:

Author’s address: Victor Yodaiken, vy@e27182.com, Independent Researcher, 2718 Creeks Edge Parkway, Austin, Texas, USA, 78733.

- To introduce a method involving certain recursive functions for specifying and verifying large scale discrete state systems.
- To illustrate the application of the method to analysis of problems in computer systems design including distributed consensus network algorithms and real-time.
- To precisely define the class of recursive functions of interest and show how they relate to classical state machines, automata products, and algebraic automata theory.

The problem addressed here is not how to validate code, but how to understand and validate *designs* prior to implementation and test. The motivation comes out of the author's experience designing, coding, and managing design and development projects for operating systems, real-time, and other "systems" [BBG83, YB97, DMY99, DY16] without any satisfactory method for even specifying design goals[1]. The solution proposed here is based on Moore type automata - deterministic state machines with an output associated with each state [HU79, Moo64] . Since the 1960s, it has been known that digital systems can be faithfully described by state machines and interconnected digital systems by state machine products, but that the standard methods for representing state machines do not scale well. This paper shows how state machines can be defined via a certain class of recursive functions so that for a finite sequence of events, w, $f_M(w)$ will be the output of state machine M in the state M reaches by following w from its initial state (see figure 1).

$$\text{Input sequence } w \longrightarrow \boxed{\text{Moore Machine } M} \longrightarrow \text{Output } = f_M(w)$$

Fig. 1. Sequence maps representing state machines

Function composition can be used to build up more complex state system descriptions from simpler ones and also, perhaps unexpectedly, to describe interconnected composite systems with components that change state in parallel. This approach permits concise but exact definitions of state machines and abstract properties of state machines that are completely impractical for state diagrams or transition maps where state sets need to be enumerated. In particular, state machines constructed with general automata products [Har64, Géc86], in which arbitrary communication can be specified via "feedback", can be defined in terms of function composition (see figure 2).

In the hopes that the basic ideas are reasonably intuitive, section 2 is an attempt to explain the method via examples and section 3 specifies and validates the notoriously slippery Paxos distributed consensus algorithm[Lam01] as an indication that the methods can be applied to substantial problems. Section 4 defines the class of primitive recursive sequence functions and shows precisely how they relate to Moore type automata and Moore machine products. Readers who are skeptical of the appeals to intuition in sections 2

[1] Such a judgment is necessarily at least somewhat subjective. See section 5 for more discussion.

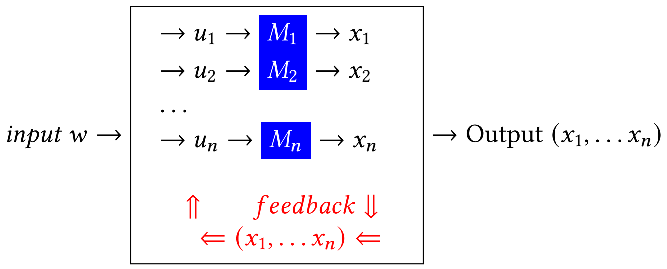
input $w \longrightarrow$
$\longrightarrow u_1 \longrightarrow M_1 \longrightarrow x_1$
$\longrightarrow u_2 \longrightarrow M_2 \longrightarrow x_2$
...
$\longrightarrow u_n \longrightarrow M_n \longrightarrow x_n$
$\longrightarrow$ Output $(x_1, \dots x_n)$
$\Uparrow$ feedback $\Downarrow$
$\Leftarrow (x_1, \dots x_n) \Leftarrow$

Fig. 2. The general product

and 3 might want to read section 4 first. Section 5 discusses motivation, background, and differences from related work in both formal methods and algebraic automata theory.

2 INTRODUCTION TO APPLICATIONS AND METHOD

Maps on finite sequences[2] can be concisely defined with "primitive recursion on words" [Pet82]. Let ϵ be the empty sequence and $w \cdot a$ be the sequence obtained by appending event a to sequence w on the right. Then $f(\epsilon)$ is the output in the initial state and given some constant c and map g

$$f(\epsilon) = c, \text{ and } f(w \cdot a) = g(f(w), a)$$

defines f in every reachable state[3]. To illustrate:

$$Counter(\epsilon) = 0 \text{ and } Counter(w \cdot a) = Counter(w) + 1 \bmod k$$

counts events mod k for positive integer k. $Counter$ is really a family of finite state systems with k as a parameter. The equations $U(\epsilon) = 0$ and $U(w \cdot a) = U(w) + 1$ define an unbounded counter[4].

Partially specified sequence maps are often useful. Suppose for some constant $0 \le c < k$ and mod k counter, $Counter$,

$$|(C(w) - Counter(w))| \le c.$$

Then $C(w)$ is a counter with error of c or less. Any sequence map that is a solution to this inequality has a useful property of counting within the error ranges, but we are not required to specify exactly what the value is in any particular state. Sequence maps specify deterministic systems but systems of interest are

[2] All sequences of events in this paper are finite.

[3] More precisely, suppose we have a set A of events, a set Y of outputs, some constant $c \in Y$, and $g : Y \times A \to Y$, then let $f(\epsilon) = c$ and for all $a \in A$ and $w \in A^*$, $f(w \cdot a) = g(f(w), a)$.

[4] A wary reader might note here that despite all the protestations about classical automata theory, U does not describe a *finite* state Moore machine. More on this in sections 4 and 5 , but while finiteness is a critical and necessary property of systems, sometimes we don't know or care about the bound and sometimes it's useful to have possibly unbounded imaginary systems to help measure or constrain behavior.

often not completely specified. Rather than positing some irreducible “non-determinism”, we can think of these maps as solutions to constraints which may have many different solutions.

Ordinary function composition can modify the output. For some constant c and the mod k counter, $Down(w) = k - Counter(w)$ is a down counter to 1.

Or suppose C_1 is a mod k_1 counter and C_2 is a mod k_2 counter and

$$F(w) = \begin{cases} 0 & \text{if } C_1(w) + C_2(w) = 0 \\ 1 & \text{otherwise} \end{cases}$$

Then $F(w) = 0$ only when the number of events is divisible by both k_1 and k_2. Suppose the alphabet of events is $\{0, 1\}$ and we want to count consecutive 1's with “saturation” counter

$$RCounter(\epsilon) = 0, \quad RCounter(w \cdot a) = \begin{cases} 0 & \text{if } a = 0; \\ RCounter(w) + 1 & \text{if } a = 1 \text{ and } RCounter(w) < k \\ Rcounter(w) & \text{otherwise.} \end{cases}$$

$RCounter(w) = k$ if the last k or more events have been equal to1.

2.1 Shift register and connection maps

In addition to modifying the output, with composition, $h(f(w))$, it is possible to modify the input by making the event sequence a dependent variable where $u(w)$ translates w to a different event sequence and $f(u(w))$ is the output of f in the state determined by $u(w)$. This is how networks and any other interconnected systems in the examples are specified. To get an intuition of how connections work, consider an elementary example: an idealized shift register constructed by connecting storage cells. A storage cell is defined trivially by

$$Cell(w \cdot a) = a \tag{1}$$

to just remember the most recent event. The cells are connected in to each other as in figure 3 by connection maps u_i defined below so that $Cell(u_i(w))$ is the value stored in the i^{th} cell.

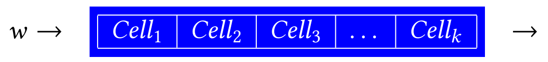

Fig. 3. Shift register

Define

$$u_i(\epsilon) = \epsilon, \quad u_i(w \cdot a) = \begin{cases} u_i(w) \cdot a & \text{if } i = 1 \\ u_i(w) \cdot Cell(u_{i-1}(w)) & \text{if } 1 < i \leq k \end{cases}$$

Consider $Cell(u_1(w))$. The initial value is unspecified so $Cell(u_1(\epsilon)) = Cell(\epsilon)$ is undefined. Suppose the first event is 1, then $Cell(u_1(\epsilon \cdot 1)) = 1$ and $Cell(u_2(\epsilon \cdot 1 \cdot 2)) = 1$ and $Cell(u_1(\epsilon \cdot 1 \cdot 2)) = 2$.

The connection maps follow a standard scheme $u_d(\epsilon) = \epsilon$ so the components all start in their initial states, and either $u_d(w \cdot a) = u_d(w) \cdot b$ for some event b in the event alphabet of the component, or $u_d(w \cdot a) = u_d(w)$ to leave the component state unchanged [5].

There's a definition of a memory array in appendix A.1.

2.2 A controller for a physical plant

Consider a control system where inputs are samples of some signal. A simple controller that produces a control signal, might have a fixed set point κ_0 and an error:

$$E(\epsilon) = 0 \text{ and } E(w \cdot y) = \kappa_0 - y$$

where y is a signal sample. The change in the last error signal (the "derivative") is

$$D(\epsilon) = 0 \text{ and } \quad D(w \cdot y) = E(w) - (\kappa_0 - y).$$

The sum of all errors is:

$$S(\epsilon) = 0 \text{ and } \quad S(w \cdot y) = S(w) + (\kappa_0 - y)$$

although we would probably want, at some point of development, to force a bound on this sum or to show that it is bounded by some external condition such as an oscillation around the x-axis. Then the control signal might be calculated using three parameters as follows.

$$Control(w) = \kappa_1 E(w) + \kappa_2 D(w) + \kappa_3 S(w). \tag{2}$$

Let

$$v(\epsilon) = \epsilon,\ v(w \cdot a) = \begin{cases} v(w) \cdot 1 & \text{if } E(w) > \kappa_4 \\ v(w) \cdot 0 & \text{otherwise} \end{cases}$$

Here '1' and '0' are treated as event symbols. $RCounter(v(w)) = k$ if and only if the error has been greater than κ_4 for at least the last k time units.

If there is a physics model of the plant, then $U(w)$ should approximate the real valued time variable. For example, we could require that if t is a real-valued time variable in seconds that $U(w) \approx t$, perhaps so that it is within 10 picoseconds or whatever level of precision is appropriate.

Imagine there are n control systems embedded within a system so that events are vectors $\vec{x} = (x_1, \dots x_n)$ where $\vec{x}_i = x_i$. A map $u_i(w)$ can isolate the inputs on line i as follows:

$$u_i(\epsilon) = \epsilon, \quad u_i(w \cdot \vec{x}) = u_i(w) \cdot \vec{x}_i$$

[5] Where useful, the components can be defined to advance by multiple steps on a single event of the enclosing system. See section 4.1.3

Then $Control_i(w) = Control(u_i(w))$ is a controller that operates as specified by equation 2 but on the indicated signal. This is the simplest parallel system- with no interaction between components (see figure 4).

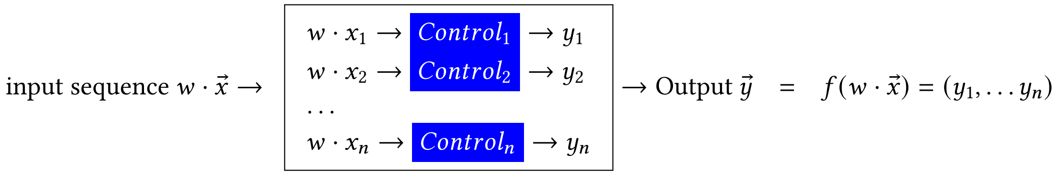
input sequence $w \cdot \vec{x} \longrightarrow$
$w \cdot x_1 \longrightarrow$
$Control_1$
$\longrightarrow y_1$
$w \cdot x_2 \longrightarrow$
$Control_2$
$\longrightarrow y_2$
...
$w \cdot x_n \longrightarrow$
$Control_n$
$\longrightarrow y_n$
$\longrightarrow$ Output $\vec{y} = f(w \cdot \vec{x}) = (y_1, \dots y_n)$

Fig. 4. Direct product of sequence maps

Suppose, on the other hand, that two controllers, say, p and s, are interconnected so that the input signal to s (secondary) should be the output of controller p (primary) and the inputs to controller p should be the pair $(\vec{x}_p, y)$ where $\vec{x}_p$ is the signal from the "plant" and y is the output from controller s. Then the connector map can be given:

$$u_i(\epsilon) = \epsilon, \quad u_i(w \cdot \vec{x}) = \begin{cases} u_i(w) \cdot Control_p(w) & \text{if } i = s \\ u_i(w) \cdot (\vec{x}_p, Control_s(w)) & \text{if } i = p \\ u_i(w) \cdot \vec{x}_i & \text{otherwise} \end{cases} .$$

where $u_s(w) \cdot Control_p(w)$ appends the output of controller p in the state determined by w as an event symbol to the sequence $u_s(w)$.

2.3 A system of interconnected processes

Consider a system of connected processes with synchronous message passing. The specification can be split into a specification of processes $P : A^* \longrightarrow X$ for process event alphabet A, followed by a specification of the interconnects. Suppose there is a set I of process identifiers and each process has output either $(read, j)$ to

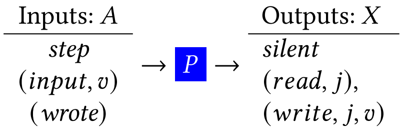

Fig. 5. An abstract process

request a message from process j, or $(write, j, v)$ to send value v to process j, or $silent$ to do some internal processing with no I/O in this state. The events then are pairs $(input, v)$ to receive value v, $wrote$ for success in sending a message, and $step$ for some internal computation. Each of these is a discrete event symbol, not an expression of any sort (see figure 5).

Suppose for each $i \in I$, P_i is a process. The processes could all be identical or could be different. Connection maps $u_i : B^* \longrightarrow A^*$ for composite system event alphabet B determine how the processes are connected. Then $P_i(u_i(w))$ is the output of process i in the state determined by $u_i(w)$.

$u_i(\epsilon) = \epsilon$ and

$$u_i(w \cdot b) = \begin{cases} u_i(w) \cdot \mathit{step} & \text{if } P_i(u_i(w)) = \mathit{silent} \\ u_i(w) \cdot (input, v) & \text{if } P_i(u_i(w)) = (read, j) \text{ for some } j \\ & \text{and } P_j(u_j(w)) = (write, i, v) \\ u_i(w) \cdot \mathit{wrote} & \text{if } P_i(u_x(w)) = (write, j, v) \text{ for some } v \\ & \text{and } P_j(u_j(w)) = (read, i) \\ u_i(w) & \text{otherwise} \end{cases}$$

This specification provides synchronous communication so that a process waiting for a message from process j or trying to send a message to process j will make no progress until process j has a matching output - if ever. Such systems are known for their propensity to deadlock: e.g. if $P_i(u_i(w)) = (read, j)$ and $P_j(u_j(w)) = (read, i)$ or if larger cycles are created. Some solutions require strict ordering of interactions, but it's more common to add timeouts. Suppose we have a map $\tau : B \longrightarrow \mathbb{R}$ associating each discrete system event in B with a duration in seconds. Define $\mathit{Blocked}_i(\epsilon) = 0$ and

$$\mathit{Blocked}_i(w \cdot b) = \begin{cases} \tau(b) + \mathit{Blocked}_i(w) & \text{if } P_i(u_i(w)) = (write, j, v) \text{ for some } j \\ & \ \text{and } P_j(u_j(w)) \neq (read, i) \\ & \text{or } P_i(u_i(w))) = (read, j) \\ & \ \text{and } P_j(u_j(w)) \neq (write, i, v) \text{ for any } v \\ 0 & \text{otherwise} \end{cases}$$

Then we could add a new *timeout* event to A, a constant κ and an additional case to u_i where $u_i(w \cdot b) = u_i(w) \cdot \mathit{timeout}$ if $\mathit{Blocked}_i(w) > \kappa$. By way of contrast, the network in section 3 is completely asynchronous.

The specification so far is completely parallel and synchronous - each process advances by one step on each network step if it can. One alternative would be to add a scheduler so only scheduled tasks advance (see appendix A.2 for details).

3 THE PAXOS PROTOCOL

The first part of this section is a general packet network connecting network agents. The second part adds constraints to the agents so they obey the Paxos protocol. The goal is to supplement the exposition in “Paxos Made Simple” [Lam01].

3.1 Packet network

Consider a networked agent which could be a device, a program, or an operating system, and only changes state when it receives or sends a message or does some internal computation. Each agent that will be connected on the network has a unique identifier. There is a set of messages *Messages*, a set of identifiers *Ids* and a map $source : Messages \longrightarrow Ids$ which is intended to tag a message with the identifier of the agent that sent it.

Definition 3.1. *$G : Messages^* \longrightarrow Messages$ is a network agent with identifier $i \in Ids$ only if:*

$$\textit{If}\ G(w) \neq 0\ \textit{then}\ source(G(w)) = i$$

(it correctly labels the source of messages it sends.)

The null message $0 \in Messages$ is used both for agent output to indicate the agent has nothing to send and as an input to indicate the network has nothing to deliver.

Definition 3.2. *The cumulative set of messages received in the state determined by $q \in Messages^*$ is given by:*

$$Received(\epsilon) = \emptyset, \quad Received(q \cdot m) = \begin{cases} Received(q) \cup \{m\} & \textit{if}\ m \neq 0 \\ Received(q) & \textit{otherwise} \end{cases}$$

The null message is ignored.

Definition 3.3. *The cumulative set of messages transmitted by G in the state determined by q is given by:*

$$Sent(G, \epsilon) = \emptyset, \quad Sent(G, q \cdot m) = \begin{cases} Sent(G, q) \cup \{G(q)\} & \textit{if}\ G(q) \neq 0 \\ Sent(G, q) & \textit{otherwise} \end{cases}$$

If $G(q) = 0$ then $Sent(G, q \cdot m)$ is unchanged. If G and G' are both agents, there is no requirement that $Sent(G, w) = Sent(G', w)$ even if both have the same identifier.

Note that once a message is in $Received(q)$ or $Sent(G, q)$ it’s there forever - new messages can put a new element in one of those sets but cannot remove any messages.

Lemma 1. *If G is a network agent with identifier i and $m \in Sent(G, q)$ then $source(m) = i$.*

The proof is simple, but the same inductive proof method is used here multiple times so it's worth going into a bit of detail. By the definition of *Sent*, $Sent(G, \epsilon) = \emptyset$ so the claim is trivially true for a sequence of length 0. Suppose the claim is true for q. If $m \in Sent(G, q)$ then $source(m)$ doesn't change with any event so the claim is true for $q \cdot a$. If $m \notin Sent(G, q \cdot a)$ the claim is also trivially true. But if $m \notin Sent(G, q)$ and $m \in Sent(G, q \cdot a)$ then, by the definition of *Sent*, $G(q) = m \neq 0$ which, by definition 3.1 where G has identifier i requires that $source(G(q)) = i$.

Definition 3.4. *A* standard packet network *consists of a collection of agents, $G_i : i \in Ids$ where each G_i has identifier i, plus connection maps*

$$u_i : B^* \longrightarrow Messages^*$$

where B is the network event alphabet and

(1) Every agent begins in the initial state and each system event causes one or zero events for each agent: $u_i(\epsilon) = \epsilon$ *and* $u_i(w \cdot a) = u_i(w)$ *or* $u_i(w \cdot a) = u_i(w) \cdot m$ *for some* $m \in Messages$.

(2) A non-zero message can only be delivered if it was previously sent: $u_i(w \cdot b) = u_i(w) \cdot m$ *for* $m \neq 0$ *only if for some* j*,*$m \in Sent(G_j, u_j(w))$.

Lemma 2. *If* $m \in Received(u_i(w))$ *then* $m \in Received(u_i(w \cdot a))$ *and if* $m \in Sent(G_i, u_i(w))$ *then* $m \in Sent(G_i, u_i(w \cdot a))$

The proof follows immediately from the definitions of txd, rxd, and from the standard u_i property that $u_i(w)$ must be a prefix of $u_i(w \cdot b)$.

Lemma 3. $m \in Received(u_i(w))$ *only if for* $j = source(m)$, $m \in Sent(G_j, u_j(w))$

Proof: As usual, this is trivially true for ϵ. Suppose $m \notin Received(u_i(w))$ but $m \in Received(u_i(w \cdot b))$. Then by the definition of *Received* and u_i we know that $u_i(w \cdot b) = u_i(w) \cdot m$ and by definition 3.4 $m \in Sent(G_j, u_j(w))$ for some j. Lemma 1 completes the proof.

The network can lose or reorder messages, but can never deliver spurious messages. In the sequel, q will always be used as a variable over $Messages^*$ and w over network sequences. Properties of agents that are local, that are true for all q, do not depend on the network interconnect.

This network is not all that much - in fact, it is almost exactly the network that was specified for a famous impossibility "theorem" [LM86] that says there is no algorithm for detecting whether an agent has failed or is just slow to respond from its inputs and outputs. The trick is that, so far, there is no notion of time for agents, so arbitrary pauses are possible and not detectable.

3.2 Paxos

Paxos[Lam01] is a 2 phase commit protocol with a twist[GL04]. The protocol relies on two sets of agents, a set P of proposers and a set C of acceptors. There are 4 non-zero message types used in the protocol and type 0 indicates the message is not part of the protocol:

$$PTypes = \{Prepare, PrepareAck, Proposal, ProposalAck, 0\}.$$

A "proposer" agent first requests permission to use a sequence number by sending a "prepare" message that carries a sequence number. If and when a quorum of acceptors agree by sending back "prepare accept" messages with the sequence number, the proposer can send a "proposal" with a proposal value and the same sequence number. When or if a majority of acceptors agree to the proposal by sending proposal accept messages, the proposal has "won". The 2 phase twist is that during the prepare accept phase, the proposer can be forced to adopt a proposal value already tried by some lower numbered proposal. This is the most complex part of the protocol (see rule 3j and 2e below) and it produces a result that multiple proposers can "win" a consensus, but they must all end up using the same value.

A *Paxos group* consists of a standard network, two subsets of ids P (proposers) and C (acceptors), a constant $\kappa > 0$ that is more than half of the number of elements of C (a quorum count), and a set $Pmsgs \subset Messages$ of messages used in the protocol. There is a map $T : Messages \rightarrow Ptypes$ to identify the Paxos message type of every message. Each Paxos message has a sequence number $seq : Pmsgs \rightarrow \mathbb{N}$. Proposal messages have a value, if $T(p) = Proposal$ then $val(p)$ is defined. Prepare accept messages may carry a proposal message $p = prev(m)$ which is the highest numbered proposal that the acceptor had accepted when it sent the prepare accept. $Prev(m) = 0$ if the acceptor had not accepted any proposals. This will be made precise below. Finally, a map $\pi : \mathbb{N} \rightarrow P$ partitions sequence numbers so that no two proposers are associated with the same sequence number.

None of the constants or maps above depend on state but the process of controlling proposal values involves two state dependent maps that are both "local" in the sense that they do not depend on network connectivity. $Accepted(i, q)$ is the set of proposal messages site i has received and then accepted by transmitting a proposal accept message with a matching sequence number.

$$Accepted(i, q) = \left\{ \begin{array}{l} p \in Received(q) : T(p) = Proposal \\ \text{and for some } m \in Sent(G_i, q), T(m) = ProposalAck \\ \text{and } seq(m) = seq(p) \end{array} \right\}$$

$Inherit(n, q)$ is the proposal with the highest sequence number that has been carried in the $prev$ attribute in a received prepare accept message with sequence number n.

$$Inherit(n, \epsilon) = 0$$

$$\mathit{Inherit}(n, q \cdot m) = \begin{cases} \mathit{prev}(m) & \text{if } T(m) = \mathit{PrepareAck} \\ & \quad \text{and } \mathit{seq}(m) = n \text{ and } \mathit{prev}(m) \neq 0 \\ & \quad \text{and either } (\mathit{Inherit}(n, q) = 0 \\ & \qquad \text{or } \mathit{seq}(\mathit{Inherit}(n, q)) < \mathit{seq}(\mathit{prev}(m))) \\ \mathit{Inherit}(n, q) & \text{otherwise} \end{cases}$$

The Paxos algorithm can be expressed in 4 rules that control when an agent can transmit a Paxos message of each non-zero type. All four rules are local to agents.

(1) $T(G_i(q)) = \mathit{Prepare}$ only if

(a) $i \in P$

(b) and $\mathit{seq}(G_i(q)) > 0$ and $\pi(\mathit{seq}(G_i(q)) = i$

(c) and there is no $m \in \mathit{Sent}(G_i, q)$ with $T(m) = \mathit{Prepare}$ and $\mathit{seq}(m) > \mathit{seq}(G_i(q))$.

(2) $T(G_i(q)) = \mathit{PrepareAck}$ only if

(a) $i \in C$

(b) and there is some $m \in \mathit{Received}(q)$ with $T(m) = \mathit{Prepare}$ and $\mathit{seq}(m) = \mathit{seq}(G_i(q))$

(c) and there is no $m \in \mathit{Sent}(G_i, q)$, where $T(m) = \mathit{PrepareAck}$ and $\mathit{seq}(m) > \mathit{seq}(G_i(q))$.

(d) and there is no $m \in \mathit{Sent}(G_i, q)$, where $T(m) = \mathit{ProposalAck}$ and $\mathit{seq}(m) \geq \mathit{seq}(G_i(q))$.

(e) and either $\mathit{Accepted}(i, q) = \emptyset$ and $\mathit{prev}(G_i(m)) = 0)$ or $\mathit{prev}(G_i(q))$ is the element of $\mathit{Accepted}(i, q)$ with the highest sequence number).

(3) $T(G_i(q)) = \mathit{Proposal}$ only if

(f) $i \in P$

(g) and there is some $m \in \mathit{Sent}(G_i, q))$, $T(m) = \mathit{Prepare}$ and $\mathit{seq}(m) > \mathit{seq}(G_i(m))$

(h) and there is no $p \in \mathit{Sent}(G_i, q)$ with $T(p) = \mathit{Proposal}$ and $\mathit{seq}(p) \geq \mathit{seq}(G_i(q))$.

(i) and $\{\mathit{source}(m) : m \in \mathit{Received}(q), T(m) = \mathit{PrepareAck}$ and $\mathit{seq}(m) = \mathit{seq}(G_i(q))\}$ has κ or more elements.

(j) and if either $\mathit{Inherit}(\mathit{seq}(T_i(q)), q) = 0$ or $(\mathit{Inherit}(\mathit{seq}(T_i(q)), q) \neq 0$ and $\mathit{val}(G_i(q)) = \mathit{val}(\mathit{Inherit}(\mathit{seq}(T_i(q)), q))$

(4) $T(G_i(q)) = \mathit{ProposalAck}$ only if

(a) $i \in C$

(b) and there is some $p \in \mathit{Received}(q)$ with $T(p) = \mathit{Proposal}, \mathit{seq}(p) = \mathit{seq}(G_i(q))$

(c) and there is no $m \in \mathit{Sent}(G_i, q)$ where $T(m) \in \{\mathit{ProposalAck}, \mathit{PrepareAck}\}$ and $\mathit{seq}(m) > \mathit{seq}(G_i(q)$.

A proposal message p “wins” in the state determined by sequence q if the proposing agent i has sent the proposal message and received proposal accept messages for p that have source identifiers from κ or more agents. This is a property of agents, not the network, because agents need to be able to decide if a proposal has won on the basis of local data.

Definition 3.5. *$Wins(i,q,p)$ if and only if*

$$\begin{aligned}&T(p) = \mathit{Proposal}\ \mathit{and}\ p \in \mathit{Sent}(G_i, q)\\&\mathit{and}\ A(p,q) = \{source(m) : m \in \mathit{Received}(q), T(m) = \mathit{ProposalAck}\ \mathit{and}\ \mathit{seq}(m) = \mathit{seq}(p)\}\\&\ \mathit{has}\ \kappa\ \mathit{or\ more\ elements.}\end{aligned}$$

Suppose $Wins(i, u_i(w), p)$ is true. By lemma 3 each propose accept message m received by agent i must have been sent by $source(m)$ so κ or more agents sent propose accept messages to agent i. By rule 4 each of those agents must be acceptors (with identifier in the set C).

This is a network property. The event sequence w determines network state and $u_i(w)$ determines the state of agent i and depends on w.

Theorem 1. *If $Wins(i, u_i(w), p)$ and $Wins(j, u_j(w), p')$ then $val(p) = val(p')$.*

3.3 Proof sketch

See appendix B for some details, but the proof can be outlined as follows. The theorem is proved by creating a list in sequence number order:

$$L(w) = p_0, p_1 \dots p_n$$

where p_0 is the winning proposal with the *least* sequence number of any winning proposal and the other elements are all the proposals so that for some j, $p \in \mathit{Sent}(G_j, u_j(w))$ and $\mathit{seq}(p) > \mathit{seq}(p_0)$. No proposals with a number less than $\mathit{seq}(p_0)$ can have won, because p_0 is picked to be the winning proposal with the least sequence number. Every winning proposal must have been sent by some G_j from the definition of $Wins$. So the list contains every winning proposal and perhaps some that didn't win but were just proposed. By rule 3h, no proposer ever sends two different proposals with the same number. By rule 3g a proposer can only send a proposal if it has previously sent a prepare proposal with the same number and by rule 1b that number must map to the id of the proposer under π so no two proposer agents can use the same proposal number. From these considerations the elements of the list never have duplicate numbers and can be strictly sorted by sequence number.

Every proposal p_x on the list with $0 < x \leq n$ must have had its prepare proposal accepted by κ or more acceptors and κ or more acceptors sent proposal accept messages for p_0 there must be at least one acceptor that has sent both kinds of messages. Suppose one of those acceptors has identifier c:

$$m_0 \in \mathit{Sent}(G_c, u_c(w)) \text{ and } m_x \in \mathit{Sent}(G_c, u_c(w))$$

where m_0 is a proposal accept message with $\mathit{seq}(m_0) = \mathit{seq}(p_0)$ and m_x is a prepare accept message with $\mathit{seq}(m_x) = \mathit{seq}(p_x)$. By rule 2d when G_c sent m_0 it could not have already sent m_x. So when $G_c(q) = p_x$ it must be that $m_0 \in \mathit{Sent}(G_c, q)$. But to send m_0, c must have received p_0 so $\mathit{Accepted}(c, q) \neq \emptyset$ and in fact the proposal with the highest sequence number in $\mathit{Accepted}(c, q)$ must have sequence number at least equal to $\mathit{seq}(p_0)$. That proposal cannot have sequence number greater than $\mathit{seq}(m_x)$ by rule 2d. In fact, the proposer of p_x can only have received prepare accept messages with sequence number less than $seq(p_x)$. It follows that when that proposer sent p_x, it was forced to adopt an inherited value from some proposal with sequence number greater than or equal to the sequence number of p_0 but less than the sequence number of p_x. Thus, the proposer of p_1 on the list must have inherited the value of p_0, the proposer of p_2 must have inherited either the value of p_0 or the value of p_1 which is the same as the value of p_0 and so on.

Every winning proposal must be on the list and must have the same proposal value so theorem 1 is proved.

3.3.1 Discussion. The specification here is compact but more detailed than the original one in [Lam01] . For example, agents are defined to output at most one message in each state — something that appears to be assumed but not stated in the original specification. If $G_i(q)$ could be a set[6], an agent could satisfy the specification and send an accept for p_0 and a prepare accept for $seq(p)$ at the same time - so that the prepare accept did not include a previous proposal with a number greater than or equal to $seq(p_0)$. Or consider what happens if an agent transmits a proposal before any proposal has won after receiving κ prepare accepts and then receives an additional prepare accept with a higher numbered prior proposal. The "Paxos made simple" specification does not account for this possibility and would possibly permit the agent to send a second proposal with a different value but the same sequence number but rule 3h forbids it here. This is the kind of detail that it's better to nail down in the specification before writing code.

The proof is intended to convey an intuition about how the protocol works, but to also be sufficiently precise considering that it relies on the states of multiple connected components that change state in parallel.

For contrast, see proofs using formal methods and proof checkers in [CLS16] and [GS21].

The proof only proves the system is "safe" (at most one value can win). Proving liveness (that some value *will* win) is a problem, because neither the network as specified (which doesn't need to ever deliver any messages) or Paxos (which can spin issuing new higher numbered proposals that block others, indefinitely) is live as specified. It would be possible to fix the specification to: rate limit the agents, require eventual delivery from the network, and put timeouts on retries (as done in some implementations [CGR07]).

4 PRIMITIVE RECURSIVE SEQUENCE FUNCTIONS AND STATE MACHINES

4.0.1 Standard Representation of Moore Machines. The standard representation of a Moore type state machine[HU79] is a sextuple:

Definition 4.1. *A* Moore machine tuple *consists of:*

$$M = (A, S, \sigma_0, X, \delta, \lambda)$$

where A is a set of discrete events (or "input alphabet"), S is a state set, $\sigma_0 \in S$ is the start state, X is a set of outputs and $\delta : S \times A \to S$ and $\lambda : S \to X$ are, respectively, the transition map and the output map.

[6] For example, if the agent is multi-threaded without appropriate locking.

The state set and alphabet are usually required to be finite, but here we sometimes don't need that restriction. The next step is to define a class of functions that are equivalent to Moore machine tuples in a strong sense.

4.0.2 Characteristic function of a Moore machine tuple. The set A^* is the set of finite sequences over the set A, including the empty sequence ϵ. For a finite sequence w and some $a \in A$ let $w \cdot a$ be the sequence obtained by appending a to w on the right.

Each Moore machine tuple is associated with a unique map $f : A^* \to X$.

Definition 4.2. *The* characteristic sequence map *of Moore machine tuple* $M = (A, S, \sigma_0, X, \delta, \lambda)$ *is :*

$$f_M(w) = \lambda(f'_M(w)) \textit{ where } f'_M(\epsilon) = \sigma_0, \quad f'_M(w \cdot a) = \delta(f'_M(w), a).$$

Distinct Moore machine tuples can have the same characteristic sequence map: for example adding non-reachable states or duplicate states or even changing the state names produces a distinct tuple. One of the advantages of working with sequence maps is that these differences are not important in the context of specifying how systems behave.

4.0.3 Sequence primitive recursion. Sequence primitive recursion [Pet82] is a generalization of arithmetic primitive recursion [Pet67] so that the set A^* takes the place of the natural numbers, the empty sequence ϵ takes the place of 0 and "+1" corresponds to $w \cdot a$.

Definition 4.3. *A map* $f : A^* \to X$ *is* sequence primitive recursive *(s.p.r) if and only if there is a set* Y*, a constant* $c \in Y$*, and maps* $h : Y \to X$ *and* $g : Y \times A \to Y$ *so that*

$$f(w) = h(f'(w))$$

where $f'(\epsilon) = c$ *and* $f'(w \cdot a) = g(f'(w), a)$ *for all* $w \in A^*$ *and* $a \in A$.

Call, (c, h, g) a *basis* for f (the set Y is implicit in the image of g).

As a very simple example from section 2 consider

$$Counter(\epsilon) = 0 \text{ and } Counter(w \cdot a) = Counter(w) + 1 \bmod k$$

Let $\iota(x) = x$ be the identity map and $g(n, a) = n + 1 \bmod k$ then $(0, \iota, g)$ is a basis for $Counter$. While this may seem trivial, section 4.1 shows how a basis provides a method for simplifying complex sequence maps and showing they are s.p.r. .

Lemma 4. *The characteristic sequence map of Moore machine tuple is s.p.r.*

For proof, take the map f_M from definition 4.2 above and let $(\sigma_0, \delta, \lambda)$ be a basis for $f(w) = \lambda(f'(w))$ where $f'(\epsilon) = \sigma_0$ and $f'(w \cdot a) = \delta(f'(w), a)$. Induction on the length of w shows $f = f_M$.

Each s.p.r. function is associated with a unique Moore machine tuple.

Definition 4.4. *If $f : A^* \longrightarrow X$ is s.p.r. with basis (c, g, h) so that $g : Y \times A \longrightarrow Y$ and $c \in Y$, then*

$$M_f = (A, Y, c, X, g, h)$$

where $X = \{h(s) : s \in Y\}$ is the (c, g, h) Moore machine tuple.

Lemma 5. *If f is s.p.r. with basis (c, g, h) and M_f is the (c, g, h) Moore machine tuple, then the characteristic map of M_f is the original s.p.r. map f.*

The proof is an immediate consequence of the definitions.

Since each Moore machine tuple is associated with a unique s.p.r. map and each s.p.r. map with a particular basis is associated with a unique Moore machine tuple, s.p.r. maps constitute an alternative representation of the same mathematical objects that Moore machine tuples represent.

4.0.4 Finite state machines.

Definition 4.5. *A sequence primitive recursive map $f : A^* \longrightarrow X$ is* finite state *only if it has some s.p.r. basis c, h, g where g has a finite image (range).*

Lemma 6. *If M is finite state then f_M is finite state and for any finite state s.p.r. map f then for some basis (c, h, g) of f, the (c, g, h) Moore machine tuple M_f is finite state.*

Proof: if M is finite state, then δ has a finite image, so f_M is finite state. Conversely, if f is finite state with basis (c, h, g) then Y is a finite set.

4.0.5 Via Myhill equivalence. Instead of using the primitive recursive reduction of a map to a state machine tuple, the connection can be made via the Myhill equivalence [MS59].

For any map: $f : A^* \longrightarrow X$, let $w \sim_f q$ if and only if $f(w \textit{ concat } z) = f(q \textit{ concat } z)$ for all $z \in A^*$. Let $[w]_f = \{z : z \sim_f w\}$ and then put $S_f = \{[w]_f : w \in A^*\}$. Make $[\epsilon]_f$ be the start state and $\delta([w]_f, a) = [w \cdot a]_f$ and $\lambda([w]_f) = f(w)$.

4.1 Composition

Several apparently more complex types of maps can be shown to be sequence primitive recursive.

4.1.1 Direct product. If there are sequence primitive recursive maps $f_i : A^* \longrightarrow X_i$ for $i = 1 \ldots, n$ then let:

$$f(w) = (f_1(w), \ldots f_n(w))$$

This describes a system constructed by connecting multiple components that change state in parallel, without communicating as shown in figure 6.

Claim: f is sequence primitive recursive. Proof: Let (c_i, g_i, h_i) be a basis for each f_i so that $f_i(w) = h_i(f_i'(w))$, $f_i'(\epsilon) = c_i$ and $f_i'(w \cdot a) = g_i(f_i'(w), a)$ and $g_i : Y_i \times A \longrightarrow Y_i$.

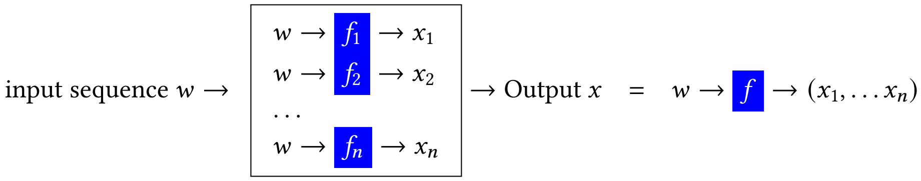
input sequence $w \longrightarrow$
$w \longrightarrow f_1 \longrightarrow x_1$
$w \longrightarrow f_2 \longrightarrow x_2$
$\dots$
$w \longrightarrow f_n \longrightarrow x_n$
$\longrightarrow$ Output x
$=$
$w \longrightarrow f \longrightarrow (x_1, \dots x_n)$

Fig. 6. Direct product of sequence maps

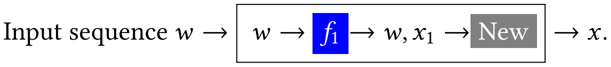
Input sequence $w \longrightarrow$
$w \longrightarrow f_1 \longrightarrow w, x_1 \longrightarrow$ New
$\longrightarrow x.$

Fig. 7. Embedding a map in a new map

Let $G((y_1, \dots y_n), a) = (g_1(y_1, a), \dots g_n(y_n, a))$ and $c = (c_1, \dots c_n)$. Let $H((y_1, \dots y_n)) = (h_1(y_1), \dots h_n(y_n))$. Then let $r(\epsilon) = c$ and $r(w \cdot a) = G(r(w), a)$. Clearly r is s.p.r.. Now we prove $r(w) = (f_1'(w), \dots f_n'(w))$ by induction on w.

$$\begin{aligned}
&r(\epsilon) = (c_1, \dots c_n) = (f_1'(\epsilon), \dots f_n'(\epsilon))\\
&\text{Suppose } r(w) = (f_1'(w), \dots f_n'(w))\\
&\text{then } r(w \cdot a) = G(r(w), a)\\
&= G((f_1'(w), \dots f_n'(w)), a)\\
&= (g_1(f_1'(w), a), \dots g_n(f_n'(w), a))\\
&= (f_1'(w \cdot a), \dots f_n'(w \cdot a))
\end{aligned}$$

Then $H(r(w)) = f(w)$ so f is s.p.r.

4.1.2 Embedding. See figure 7. If f_1 is s.p.r., let $f(\epsilon) = \kappa$ and $f(w \cdot a) = g((f(w), f_1(w)), a)$ for s.p.r. map f_1. Claim: f is s.p.r.. Here a new p.r. sequence map is being constructed to depend both on w and on the values of $f_1(w)$. Proof. There must be a primitive recursive basis (c_1, g_1, h_1) for f_1 so that $f_1(w) = h_1(f_1'(w))$ where $f_1'(\epsilon) = c_1$ and $f_1'(w \cdot a) = g_1(f_1'(w), a)$

$$\begin{aligned}
&\text{Let } H((x, y)) = x\\
&G((x, y), a) = (g((x, h_1(y)), a), g_1(y, a))\\
&r(\epsilon) = (\kappa, c_1) \text{ and } r(w \cdot a) = G(r(w), a)
\end{aligned}$$

Clearly, r is s.p.r. Claim: $r(w) = (f(w), f_1'(w))$. In that case $H(r(w)) = f(w)$ which proves f is s.p.r.

$$\begin{array}{l}
r(\epsilon) = (\kappa, c_1) = (f(\epsilon), f_1'(\epsilon)). \\
\text{suppose } r(w) = (f(w), f_1'(w)) \text{ then} \\
r(w \cdot a) = G(r(w), a) = G((f(w), f_1'(w)), a) \\
= (g((f(w), h_1(f_1'(w))), a), g_1(f_1'(w), a)) \\
= (g((f(w), f_1(w)), a), f_1'(w \cdot a)) \\
= (f(w \cdot a), f_1'(w \cdot a)) \\
\text{End proof}
\end{array} \tag{3}$$

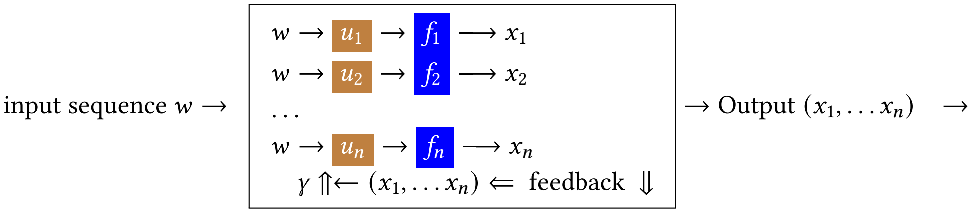
input sequence $w \rightarrow$
$w \rightarrow$ u_1 $\rightarrow$ f_1 $\longrightarrow x_1$
$w \rightarrow$ u_2 $\rightarrow$ f_2 $\longrightarrow x_2$
...
$w \rightarrow$ u_n $\rightarrow$ f_n $\longrightarrow x_n$
$\gamma \Uparrow \leftarrow (x_1, \dots x_n) \Leftarrow$ feedback $\Downarrow$
$\rightarrow$ Output $(x_1, \dots x_n)$ $\rightarrow$

Fig. 8. Event and output flow in the general product

4.1.3 General product.

Definition 4.6. *For $f_i : A_i^* \rightarrow X_i$ and $\gamma_i : X_1 \times \dots X_n \times A \rightarrow A_i^*$ where $(0 < i \leq n)$ and each f_i is s.p.r., the* general product *is:*

$$\begin{array}{l}
f(w) = (f_1(u_1(w)), \dots f_n(u_n(w))) \\
u_i(\epsilon) = \epsilon, \ \textit{and}\ u_i(w \cdot a) = u_i(w)\ \textit{concat}\ \gamma_i(f(w), a)\ \textit{for}\ i = 1, \dots n
\end{array}$$

where concat is the usual concatenation of finite sequences.

As illustrated in figure 8, if we have n p.r. sequence maps $f_1, \dots f_n$ with each $f_i : Ai^* \rightarrow X_i$, the system output when they are connected is in the set $X = X_1 \cdots \times X_n$, and a "connector" is a map $\gamma_i : A \times X \rightarrow A_i^*$ so that $\gamma_i(a, x)$ is the sequence of events produced for component i when the system input is a and the outputs of all the components are given by x. The event alphabets of the components can all be different or the same and the composite alphabet can also be different or the same depending only on γ_i.

Theorem 2. *If $f_1, \dots f_n$ are s.p.r. in a product of the type of definition 4.6, then f is s.p.r.*

Proof: Each f_i has a basis (c_i, g_i, h_i) with $g_i : Y_i \times A_i \rightarrow Y_i$ and $f_i(q) = h_i(f_i'(q))$ where

$$f_i' : A_i^* \rightarrow Y_i \text{ and } f_i'(\epsilon) = c_i \text{ and } f_i'(w \cdot a) = g_i(f_i'(w), a)$$

Let $H(y_1, \ldots y_n) = (h_1(y_1), \ldots h_n(y_n))$ so $f(w) = H(f_1'(u_1(w)), \ldots f_n'(u_n(w)))$. The goal is to define a s.p.r. map

$$r : A^* \to Y_1 \times \ldots Y_n$$

so that $r(w) = (f_1'(u_1(w)), \ldots f_n'(u_n(w)))$, which implies $H(r(w)) = f(w)$. This will prove f is s.p.r..

Because γ_i is sequence valued it is useful to extend each g_i to sequences:

$$g_i^* : Y_i \times A^*. \text{ Let } g_i^*(y, \epsilon) = y \text{ and } g_i^*(y, q \cdot a) = g_i(g_i^*(y, q), a).$$

Then let:

$$F_i'(\epsilon) = c_i \text{ and } F_i'(q \cdot a) = g_i^*(F_i'(q), \epsilon \cdot a).$$

Clearly $F_i'(q) = f_i'(q)$ so $f_i(q) = h_i(F_i'(q))$.

$$\begin{array}{l}
\text{for } y = (y_1, \ldots y_n) \text{ let } G(y, a) = (g_1^*(y_1, \gamma_1(H(y), \epsilon \cdot a)), \ldots g_n^*(y_n, \gamma_n(H(y), \epsilon \cdot a))) \\
\text{Let } r(\epsilon) = (c_1, \ldots c_n), \text{ and } r(w \cdot a) = G(r(w), a) \\
\text{By construction } r \text{ is s.p.r.} \\
\text{Claim } r(w) = (F_1'(u_1(w)), \ldots F_n'(u_n(w))) \\
\text{Proof by induction on } w \\
r(\epsilon) = (c_1, \ldots c_n) \\
= (F_1'(\epsilon), \ldots F_n'(\epsilon) \\
= (F_1'(u_1(\epsilon)), \ldots F_n'(u_n(\epsilon)) \\
\text{Inductive hypothesis } r(w) = y = (F_1'(u_1(w)), \ldots F_n'(u_n(w))) \\
r(w \cdot a) = G(r(w), a) \\
= (g_1^*(y_1, \gamma_1(H(y), \epsilon \cdot a)), \ldots g_n^*(y_n, \gamma_n(H(y), \epsilon \cdot a))) \\
= (g_1^*(F_1'(u_1(w), \gamma_1(f(w), a)), \ldots g_n^*(F_n'(w), \gamma_n(f(w), a))) \\
= (F_1'(u_1(w \cdot a)), \ldots F_n'(w \cdot a)))) \\
\mathit{QED}
\end{array} \tag{4}$$

The proof here is not complicated, but I originally produced it by going via the Moore machine tuple representation covered in section 4.2 as it was easier to visualize. In that proof, first each component map is converted to a Moore machine tuple and then they are multiplied out in the general product and then the result is converted back to the characteristic s.p.r. map.

4.2 Moore machine products

The general product of Moore machines [Har64] and later [Géc86, Yod91b] has a state set constructed as the cross product of the state sets of the factor machine and has a connector map for each component $\phi_i : X_1 \cdots \times X_n \times A \to A_i$. Compare to definition 4.6.

Definition 4.7. *Given a set A of product machine events, Moore machine tuples $M_i = (A_i, S_i, \sigma_{i,0}, \delta_i, X_i, \lambda_i)$ and connectors $\phi_i : X \times A \to A_i$ for $i = 1, \ldots .n$,* the general product:

$$M = (A, S, \sigma_0, \delta, X, \lambda) \text{ where}$$

$X = X_1 \times \ldots X_n$, and $S = S_1 \times \ldots S_n$
the initial state $\sigma_0 = (\sigma_{1,0}, \ldots \sigma_{n,0})$
the output map $\lambda((s_1, \ldots s_n)) = (\lambda_1(s_1), \ldots \lambda_n(s_n))$
and the transition map $\delta((\sigma_1, \ldots \sigma_n), a) = (\delta_1(\sigma_1, a_1), \ldots \delta_n(\sigma_n, a_n))$ where $a_i = \phi_i(\lambda_1(\sigma_1), \ldots \lambda_n(\sigma_n)), a)$

If each M_i is finite state, the product is finite state.

The connectors can be extended to produce sequences on each step just as with the general product of sequence primitive recursive functions.

Since M is a Moore type state machine tuple, it has a characteristic map. If f_i is a characteristic map for each M_i then

$$f(w) = (f_1(u_1(w)), \ldots f_n(u_n(w)))$$

where each $u_i(\epsilon) = \epsilon$ and $u_i(w \cdot a) = u_i(w) \cdot \phi_i(f_1(u_1(w)), \ldots f_n(u_n(w)), a))$ is the characteristic map for M.

4.3 Algebra

One of the motivations in this work for staying within the bounds of classical deterministic state machines is that they have been shown to be fundamental mathematical objects directly connected to semigroup and group theory[Pin86, Hol83, Gin68, BGSS91]. Each state machine defines a semigroup via congruence classes on finite sequences as shown by Nerode and Myhill (p. 70-72 [MS59]). Earlier work in algebraic automatic theory looked at connections between the component structure of computer systems and the "loop-free" product structure of the characteristic monoids. In the construction of general product of s.p.r. maps, if each γ_i depends only on the last argument, then the product reduces to a "direct" or "cross" product and the state machines are not interconnected. If each $\gamma_i(x_1, \ldots x_n, a)$ depends only on a and $x_1, \ldots x_{i-1}$ then the product reduces to a "loop free" or "cascade" product [HS66, Hol83, Pin86, Mal10] in which information only flows in a linearly ordered pipeline through the factors. But the network in section 3 is an easy example of a system where a loop-free decomposition will not reflect system architecture.

4.3.1 Modularity. Moore machines distinguish between internal state and externally visible state (output) in a way that corresponds to "information hiding" [Par72]. This is why the connectors of the general product 4.6 depend on the outputs of the components, not on their interior state sets. The product structure of systems can then provide an insight into modularity in terms of how much of the internal state of components must be communicated to other components. It is relatively simple to show that any finite state map with n states can be constructed from $\log_2 n$ single bit state maps - but at the expense of making all state information visible and communicating the state of every component to every other component on each step. The ratio of the size of the output set to the size of the set $A^*/\sim_f$ indicates the extent of information hiding. That is, the benefits of modularity cannot be automatically obtained by breaking a system into components. This is a well known engineering principle [Amd13] which, perhaps, can be investigated more in terms of automata products and monoid structure. For example, the design basis of micro-kernel operating systems is appealing, but there might be fundamental mathematical reasons why obtaining high performance is so difficult[CB93].

5 RELATED WORK

5.1 Sources

Deterministic state machines[MS59], particularly finite state machines, are well known and widely used in computer science and digital circuit engineering. Moore type machines [Moo64] add a model of interaction as input/output and of modularity via the distinction between internal state and visible state (output). The *concurrent product* given by Hartmanis in 1964[Har64] and later in [Géc86] provides semantics for composition, concurrency, and encapsulation. Algebraic automata theory [Arb69, Hol83, Gin68] includes a view of automata as maps from finite sequences of events to output. These sequence maps are used in algebraic automata theory to show equivalence between machines that “do the same thing" even though they might have different state sets. Here sequence maps are used *in place* of methods like state diagrams so that state sets do not need to be made explicit and can be parameterized and so that the emphasis can be on behavior. Primitive recursive functions on words, are adapted from Rozsa Peter’s “Recursive functions in Computer Theory" [Pet82] and“Recursive Functions” [Pet67]. There is a more abstract treatment of the same functions in [EE70]. Application of primitive recursion to describe state machines was introduced by this author as a semantics for an extended temporal logic[Yod91b, Yod91a, Yod90]. As discussed in the next section, however, this paper does not use the methods of formal logics. The treatment of interconnection by dependent variables is new to this paper as is the notion of a basis for sequence primitive recursive functions.

5.2 Comparison

> *Everybody who has worked in formal logic will confirm that it is one of the technically most refractory parts of mathematics — John von Neumann[vN41]*

> *engineers in practice are not familiar with and not fond of large logical formulas that arise if an untuned logical formalism is used [Bro97]*

This project began with temporal logic [MP79, Pnu85, CAS83, Lam94, Ram83] which adds a variety of qualifiers to a first order logic in order to be able to express *when* propositions can or must become true. The current version, however, does not use any formalism in the sense of formal logics and axiomatic methods but is based on recursive functions and ordinary applied mathematics. The goal is to "formalize" system specifications in the sense of expressing them precisely and mathematically but not to "formalize" in the sense of syntactic methods employed in the foundations of mathematics[Sho67] or in the temporal logics or similar.

In TLA[Lam94] and related approaches [AL93], the "semantics" consists of infinite sequences of assignment maps, each map assigns values to variable symbols. A component is specified with a formal expression in the logic and is evaluated against the sequence of maps. Components are composed by conjunction of their specifications. Because the semantics doesn't directly support parallel state changes or composition, interleaving is imposed axiomatically. The specification of the composed system in temporal logic includes clauses that require that the variables of the specification of a component system can always be kept constant over the "next" state transition so that some other component advances in that step. But because this would allow components to never make progress, further clauses are required to impose "fairness" on the sequences. S.p.r. maps can express interleaving, scheduling, and parallel state change directly in terms of products as shown in the examples above. See appendix C for more on temporal logics and interleaving.

FOCUS [Bro97, Bro10] on the other hand, takes named typed channels for "asynchronous, buffered message exchange" as the *primitive* method of communication between components. FOCUS semantics is given as maps from and to infinite sequences of messages (or infinite sequences of finite sequences of messages) labeled with channel identifiers. Systems are specified by interface (lists of channels and channel types) and formal expressions written in second order logics. The formal variables are assigned values by the message sequences. FOCUS can describe parallel composition with feedback, but, as with TLA, composition is described by the "and" of the specifications on the components, in this case with some additional clauses for shared channel names which determine the component communication.

FOCUS maps are both elaborate and highly specific about communication, and at the same time quite general objects — far more general than state machines. For example, these maps do not necessarily have causal relationships between input and output. That is, it is possible for a specification to define a component that can respond to events that have not happened yet but that are further down the infinite sequence of input messages. Since that's not a sensible property of a computer system, the formal specifications of a system component can be be extended to impose "strong causality" axiomatically. Broy points out that a strongly causal FOCUS map "essentially defines a deterministic automaton, called Mealy machine" ([Bro10] p.6). In fact, in hopes of making FOCUS specifications more intuitive[7] , they can be combined with state diagrams, where formal predicates are attached to transitions and states can be arbitrarily complex.

There are similarities between most methods for computer system specification because the final objects being specified are the same. S.p.r. maps differ from both TLA and FOCUS because they are constructive not axiomatic or extensional. The system design expressed in s.p.r. maps determines interconnection, concurrency and encapsulation, and s.p.r. maps are causal by construction. FOCUS and TLA (and many

[7] "often properties of systems have to be written by too lengthy, unintelligible, intricate formulas."([Bro97] p.3)

similar methods) require use of formal logics because the underlying semantics is more general than state machines (and less structured than s.p.r. maps or Moore machines in product form). System specifications are, consequently, more complicated and require something like a formal logic to add structure to the semantics. Whether this is important to the system designer or not is something that has to be determined by experience.

5.3 Future work

In addition to the avenues for future research noted in section 4.3 both specification of more real-world systems, and some automation would be useful. For the first, more medium sized examples would produce empirical information on best ways of carrying out specifications and proofs and larger examples (operating systems) would test the limits of the method. For the second, anyone reading earlier versions of this paper should immediately see the advantage of programs checking for typographical errors, type agreement (e.g. "what is p_x, shouldn't it be p_i?") and even simple dimensional analysis [Mah10]. Another useful automation would analyze specification text against a proof text to see which assertions have been cited in the proof. More ambitiously, formal logic is not required for automation of mathematics, e.g. see [vzGG13] , and integrating s.p.r. maps and proofs into a computer algebra system would be interesting. Finally, some visual method, such as Statecharts [Har84] seems like it might be practical, using s.p.r. maps to resolve or evade the "extremely delicate problems" and subtle issues [HN96] of the *ad hoc* semantics.

6 ACKNOWLEDGMENTS

Thanks very much to the reviewers and particularly to the reviewing editor, Ewen Denney, for insights, comments, encouragement and suggestions. The reviewers careful analysis and close reading was enormously helpful and much appreciated.

REFERENCES

[AL93] Martin Abadi and Leslie Lamport. Composing specifications. *Transactions on Programming Languages and Systems*, 15(1), January 1993.

[Amd13] Gene M. Amdahl. Computer architecture and amdahl's law. *Computer*, 46(12):38–46, 2013.

[Arb69] Michael A Arbib. *Theories of Abstract Automata (Prentice-Hall Series in Automatic Computation).* Prentice-Hall, Inc., USA, 1969.

[BBG83] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, page 90–99, New York, NY, USA, 1983. Association for Computing Machinery.

[BGSS91] G. Baumslag, S.M. Gersten, M. Shapiro, and H. Short. Automatic groups and amalgams. *Journal of Pure and Applied Algebra*, 76(3):229–316, 1991.

[Bro97] Manfred Broy. The specification of system components by state transition diagrams. *Institut für Informatik, Technische Universität München.*, TUM-I9729, 1997.

[Bro10] Manfred Broy. A logical basis for component-oriented software and systems engineering. *Comput. J.*, 53(10):1758–1782, 2010.

[CAS83] E. M. Clarke, Emerson A., and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. In *Proceedings of the 10th Annual Symposium on Principles of Programming Languages*, pages 117–119, 1983.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. *SIGOPS Oper. Syst. Rev.*, 27(5):120–133, dec 1993.

[CGR07] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live - an engineering perspective (2006 invited talk). In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, 2007.

[CLS16] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. Formal verification of multi-paxos for distributed consensus. *CoRR*, abs/1606.01387, 2016.

[DMY99] Cort Dougan, Paul Mackerras, and Victor Yodaiken. Optimizing the idle task and other mmu tricks. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 229–237, USA, 1999. USENIX Association.

[DY16] Cort Dougan and Victor Yodaiken. Method, time consumer system, and computer program product for maintaining accurate time on an ideal clock, 8 2016. US Patents and Trademarks Office, Patent number 20160238999.

[EE70] S. Eilenberg and Calvin Elgot. *Recursiveness.* Academic Press, New York, 1970.

[Géc86] Ferenc Gécseg. *Products of Automata*, volume 7 of *EATCS Monographs on Theoretical Computer Science.* Springer, Berlin, 1986.

[Gin68] A. Ginzburg. *Algebraic theory of automata.* Academic Press, New York, 1968.

[GL04] Jim Gray and Leslie Lamport. Consensus on transaction commit. *Computing Research Repository*, cs.DC/0408036, 2004.

[GS21] Aman Goel and Karem A. Sakallah. Towards an automatic proof of lamport's paxos. *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 112–122, 2021.

[Har64] 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.

[Har84] D. Harel. Statecharts: A visual formalism for complex systems. Technical report, Weizmann Institute, 1984.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, oct 1996.

[Hol83] W.M.L. Holcombe. *Algebraic Automata Theory.* Cambridge University Press, 1983.

[HS66] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, Englewood Cliffs, N.J., 1966.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Welsey, Reading MA, 1979.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, May 1994.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[LM86] A. Lynch, N. and M. Merrrit. Introduction to the theory of nested transactions. Technical Report TR-367, Laboratory for Computer Science, MIT, 1986.

[Mah10] Sanjoy Mahajan. *Street-Fighting Mathematics: The Art of Educated Guessing and Opportunistic Problem Solving.* The MIT Press, 03 2010.

[Mal10] Oded Maler. *On the Krohn-Rhodes Cascaded Decomposition Theorem*, page 260–278. Springer-Verlag, Berlin, Heidelberg, 2010.

[Moo64] E.F. Moore, editor. *Sequential Machines: Selected Papers.* Addison-Welsey, Reading MA, 1964.

[MP79] Z. Manna and A. Pnueli. The modal logic of programs. In *Proceedings of the 6th International Colloquium on Automata, Languages, and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 385–408, New York, 1979. Springer-Verlag.

[MS59] Rabin M.O. and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2), April 1959.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[Pet67] Rozsa Peter. *Recursive functions*. Academic Press, New York, 1967.

[Pet82] Rozsa Peter. *Recursive Functions in Computer Theory*. Ellis Horwood Series in Computers and Their Applications, Chichester, 1982.

[Pin86] J.E. Pin. *Varieties of Formal Languages*. Plenum Press, New York, 1986.

[Pnu85] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of curent trends. In J.W. de Bakker, editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[Ram83] Krithivasan Ramamritham. Correctness of a distributed transaction system. *Information systems*, 8(4):309–324, 1983.

[Sho67] Joseph R Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[vN41] John von Neumann. The general and logical theory of automata. In *Cerebral Mechanisms in Behavior*, pages 1–41. Wiley, New York, NY, USA, 1941.

[vzGG13] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra:*. Modern Computer Algebra. Cambridge University Press, 2013.

[YB97] Victor Yodaiken and Michael Barabanov. Real-Time. In *USENIX 1997 Annual Technical Conference (USENIX ATC 97)*, Anaheim, CA, January 1997. USENIX Association.

[Yod90] Victor Yodaiken. The algebraic feedback product of automata. a state machine based model of concurrent systems. In *CAV (DIMACS/AMS volume)*, pages 591–614, New Brunswick, NJ, 1990. Springer Science & Business Media.

[Yod91a] Victor Yodaiken. The algebraic feedback product of automata. In *Papers from the DIMACS Workshop on Computer Aided Verification*, AMS-DIMACS Series. American Mathematical Society, 1991.

[Yod91b] Victor Yodaiken. Modal functions for concise definition of state machines and products. *Information Processing Letters*, 40(2):65–72, October 1991.

A APPENDIX: SOME MORE SMALL EXAMPLES

A.1 Memory

Using the cell defined above in section define a memory array as in figure 9. For some set D of addresses

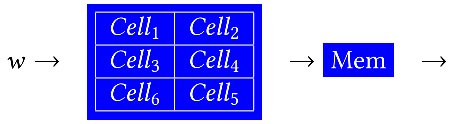

Fig. 9. Memory array

and V of values, the memory array alphabet A consists of events: $(read, d)$ to read from memory cell d, $(write, d, x)$ to write value x to cell d, and $(copy, d, d')$ to copy the contents of cell d to cell d'. All we need is one memory cell map, $Cell : V^* \longrightarrow V$ with different input sequences u_d for each $d \in D$.

$$u_d(\epsilon) = \epsilon, \quad u_d(w \cdot a) = \begin{cases} u_d(w) \cdot x & \text{if } a = (write, d, x), x \in V \\ u_d(w) \cdot Cell(u_{d'}(w)) & \text{if } a = (copy, d', d), d' \in D \\ u_d(w) & \text{otherwise} \end{cases}$$

The connection maps follow a standard scheme $u_d(\epsilon) = \epsilon$ so the components all start in their initial states, and either $u_d(w \cdot a) = u_d(w) \cdot v$ for some cell value v or $u_d(w \cdot a) = u_d(w)$ to leave the cell state unchanged [8]. In this example, the most complicated case $u_d(w) \cdot Cell(u_{d'}(w))$ appends the output of cell d' to the sequence of events being constructed for cell d.

Define:

$$Mem(w \cdot a) = \begin{cases} Cell(u_d(w)) & \text{if } a = (read, d) \text{for some } d \in D \\ Mem(w) & \text{otherwise} \end{cases}$$

to complete the system. To extend this specification to support events $(swap, d, d')$ that swap the contents of two cells, change the definition of u_d so

$$u_d(\epsilon) = \epsilon, \quad u_d(w \cdot a) = \begin{cases} u_d(w) \cdot x & \text{if } a = (write, d, x), x \in V \\ u_d(w) \cdot Cell(u_{d'}(w)) & \text{if } a = (copy, d', d), d' \in D \\ & \text{or } a = (swap, d, d') \text{ or } a = (swap, d', d), d, d' \in D \\ u_d(w) & \text{otherwise} \end{cases}$$

This is a very elementary example, but it's a definition of a deterministic state machine as a product of state machines where each component can receive input from the external system or from any other cell and cells can change state in parallel with each other and the *Mem* component.

A.2 Scheduler

There are many possible ways to add a scheduler to the system of processes of section 2.3. Perhaps on each step of the composite system the scheduler picks a set of processes to advance by one step. The inputs to the scheduler on each step might be the tuple of outputs of the process components so the input alphabet of the scheduler is $C = X^n$ where n is the number of elements in I and the outputs of the scheduler are sets of process ids to advance so a scheduler is $S : C^* \longrightarrow J$ where J contains some set of subsets of I. Define

$$v(\epsilon) = \epsilon, v(w \cdot b) = v(w) \cdot (x_1, \ldots x_n) \text{ where } x_i = P_i(u_i(w))$$

[8] Where useful, the components can be defined to advance by multiple steps on a single event of the enclosing system. See section 4.1.3

Then modify the other connectors $u_i(\epsilon) = \epsilon$ and

$$u_i(w \cdot a) = \begin{cases} u_i(w) \cdot \mathit{step} & \text{if } i \in S(v(w)) \text{ and } P_i(u_i(w)) = \mathit{silent} \\ u_i(w) \cdot (\mathit{input}, v) & \text{if } P_i(u_i(w)) = (\mathit{read}, j) \text{ for some } j) \\ & \text{and } i \in S(v(w)) \text{ and } j \in S(v(w)) \\ & \text{and } P_j(u_j(w)) = (\mathit{write}, i, v) \\ u_x(w) \cdot \mathit{wrote} & \text{if } P_i(u_x(w) = (\mathit{write}, j, v) \text{ for some } v \\ & \text{and } i \in S(v(w)) \text{ and } j \in S(v(w)) \\ & \text{and } P_j(u_j(w)) = (\mathit{read}, i) \\ u_x(w) & \text{otherwise} \end{cases}$$

B APPENDIX: SOME DETAILS OF THE PAXOS SAFETY PROOF

Lemma 7. *If an acceptor has sent both a prepare accept message and a propose accept message where the proposal accept message has a smaller sequence number, then the prepare accept message must carry a prior proposal with a sequence number at least as great as that of the proposal accept.*

If $m_x \in \mathit{Sent}(G_i, q)$ *and* $m_y \in \mathit{Sent}(G_i, q)$ *and* $T(m_x) = \mathit{ProposalAck}$ *and* $T(m_y) = \mathit{PrepareAck}$ *and* $\mathit{seq}(m_x) < \mathit{seq}(m_y)$ ***then*** $\mathit{seq}(\mathit{prev}(m_y)) \geq \mathit{seq}(m_x)$.

Proof by induction on sequence length. The lemma is trivially true for length 0 since $\mathit{Sent}(G_i, \epsilon) = \emptyset$ by definition 3.3. Assume the lemma is true for sequence z and consider $z \cdot a$. There are 4 cases for each z and a:

(1) $m_y \in \mathit{Sent}(G_i, z)$ and $m_y \in \mathit{Sent}(G_i, z)$ in which case, by the induction hypothesis $\mathit{seq}(m_x) \leq \mathit{seq}(\mathit{prev}(m_y)))$ and these are not state dependent so the inequality holds in the state determined by $z \cdot a$.

(2) $m_x \notin \mathit{Sent}(G_i, z \cdot a)$ or $m_y \notin \mathit{Sent}(G_i, z \cdot a)$ in which case the lemma is trivially true in the state determined by $z \cdot a$

(3) $m_x \notin \mathit{Sent}(G_i, z)$ or $m_y \notin \mathit{Sent}(G_i, z)$ and $m_x \in \mathit{Sent}(G_i, z \cdot a)$ or $m_y \in \mathit{Sent}(G_i, z \cdot a)$. By the definition of *Sent* one of $m \in \mathit{Sent}(G_i, z)$ or $m_y \in \mathit{Sent}(G_i, z)$ because at most one element is added to $\mathit{Sent}(G_i, z)$ by the event a. So there are two subcases:
   (a) If $m_y \in \mathit{Sent}(G_i, G_i, z)$ and $G_i(z) = m_x$ by rule 4c $T(G_i(z)) \neq \mathit{ProposalAck}$ contradicting the hypothesis. This case cannot happen.
   (b) If $m \in \mathit{Sent}(G_i, G_i, z)$ and $G_i(z) = m_y$ then, by lemma 13 there is some $p \in \mathit{Received}(q)$ so $T(p) = \mathit{Proposal}$ and $\mathit{seq}(p) = \mathit{seq}(m_x)$. The set $\mathit{Accepted}(i, q)$ in rule 2e then contains at least p, so $\mathit{seq}(\mathit{prev}(m_y)) \geq \mathit{seq}(p) = \mathit{seq}(m_x)$.

Lemma 8. *If* $\mathit{Wins}(i, u_i(w), p)$ *and* $G_j(u_j(w)) = p'$ *for some* p' *where* $T(p') = \mathit{Proposal}$ *and* $\mathit{seq}(p') > \mathit{seq}(p)$ *then there is a* $m \in \mathit{Received}(u_j(w))$, $T(m) = \mathit{PrepareAck}$ *and* $\mathit{seq}(m) = \mathit{seq}(p'$ *and* $\mathit{seq}(\mathit{prev}(p')) \geq \mathit{seq}(p)$

This is a ridiculously long chain of assertions, but almost all the work is done. From the definition of $Wins$, site i has received proposal accept messages for p from κ or more acceptors. From rule 3i agent j must have received prepare accept messages for $seq(p')$ from κ or more acceptors. Since κ is more than half, there must be at least one agent x that has transmitted both a proposal accept for p and a prepare accept m for $seq(p')$ from x. By lemma 7 then $seq(prev(m)) \geq seq(p)$.

Lemma 9. *If $m \in Sent(G_i, q)$ and ($T(m) = PrepareAck$ or $T(m) = ProposalAck$)) then $i \in C$.*
This follows directly from rules 2 and 4 and from lemma 1.

Lemma 10. *If $p \in Sent(G_j, q)$ and $T(p) = Proposal$ then $\pi(seq(p)) = j$.*

Proof: $G(q) = p$ implies, by rule 3g that there is some $m \in Sent(G_j, q)$ where $seq(m) = seq(p)$ and $T(m) = Prepare$. And rule 1b requires that $G_j(z) = m$ only if $\pi(seq(m)) = source(m)$ and lemma 1 requires that $source(m) = j$. Since $seq(m) = seq(p)$ then $source(p) = source(m) = j = \pi(source(p))$.

Lemma 11. *If $p \in Sent(G_j, q)$ where $T(p) = Proposal$ and $p' \in Sent(G_j, q)$ where $T(p') = Proposal$ and $seq(p) = seq(p')$ then $p = p'$.*

Proof: Suppose, without loss of generality that $p \in Sent(G_j, z)$ and $G_j(z) = p'$ then by rule 3h $p = p'$.

It follows that

Lemma 12. *If $p \in Sent(G_j, u_j(w))$ where $T(p) = Proposal$ and $p' \in Sent(G_i, s_i(w))$ then $j = i$ or $seq(p) \neq seq(p')$*

If $seq(p) = seq(p')$ then $\pi(seq(p)) = j = \pi(seq(p')) = i$.

Lemma 13. *If a agent has sent a proposal accept, it has received a matching proposal (with the same sequence number)*
If $m \in Sent(G_j, q), T(m) = ProposalAck$ then there is some $p, T(p) = Proposal, seq(p) = seq(m), p \in Received(q)$.

Proof by induction on prefixes of q, Initially $m \notin Sent(G_j, , \epsilon$. Suppose $m \in Sent(G_j, , z \cdot a)$ but $m \notin Sent(G_j, , z)$ then $G(z) = m$ (by definition of txd) which means by 4b the matching proposal must have been received.

Lemma 14. *If $m \in Sent(G_i, q)$ and $T(m) = PrepareAck$ then either $source(prior(m)) = 0$ or $seq(prior(m)) < seq(m))$.*

Suppose $G(q) = m$ and $T(m) = PrepareAck$. By rule 2e if $prior(m) \neq 0$ then $prior(m) = p$ so that

$$p \in \{p \in Received(q), T(p) = Proposal \text{ and } \exists m_c \in Sent(G_i, q), T(m_c) = ProposalAck, seq(m_c) = seq(p)\}$$

So $m_c \in Sent(G_i, q)$ which implies, by rule 2d that $seq(m_c) < seq(m)$.

C NOTES ON TEMPORAL LOGIC

The kinds of properties that one might express in temporal logics can also be made precise using s.p.r. maps. For example, the expression $P(w)$ where P is a boolean s.p.r. map and w is a free variable over event sequences means "P is always true". Define $NotP(w \cdot a) = (1 - P(w)) * (1 + NotP(w))$ and then $NotP(w) < t$ or "for some t, $NotP(w) < t$" are concrete versions of "eventually P". To assert that P must become true before Q becomes true, define $EdgeP(\epsilon) = 0$, $EdgeP(w \cdot a) = max(P(w \cdot a), EdgeP(w))$ then the desired property is $Q(w) \leq EdgeP(w)$. The proofs in section 3 involve showing that some condition must be true *before* a particular message can be transmitted. If we wanted to assert message delivery is "fair", we could count how many times any message has been transmitted and then add a probability measure[9].

The semantics of temporal logics come from a form of state machines where states are maps from formal variables to values [10]. The state machines or sequences are unlabeled and non-deterministic so much of behavior of the system are expressed as axioms in the logical language. Without automata products, temporal logics model composition and concurrency via interleaving. Using interleaving to model concurrency treats a concept from programming languages with concurrent threads as fundamental. Specification of an operating system, where some operations are truly parallel and some are scheduled by the OS itself does not fit into this seamlessly. In this regard see [Lam94] p.44,

> "TLA is based on an interleaving model of concurrency, in which we assume that an execution of the system consists of a sequence of atomic events. It seems paradoxical to represent concurrent systems with a formalism in which events are never concurrent. We will not attempt to justify the philosophical correctness of interleaving models for reasoning about concurrent algorithms.".

The semantics is not causal (transitions are unlabeled and considered to represent passage of a unit of time or a step). You can't say, $x(w \cdot a) = 1$ and $x(w \cdot b) = 0$, you have to say something like $next(lastevent = a \rightarrow x = 1)$ etc. and then provide additional rules for "lastevent" which can easily become quite complex. Since composition involves combining state assignment maps, any rules for "lastevent" must take into account that the other component might have different last events. or no events at all. There is no semantic connection between the assignment maps and the state nodes in the graphs — this has to be specified axiomatically. Additionally, the composed state machines are "flat" with execution interleaved non-deterministically. To express something similar to $m \in Received(u_i(w)) \rightarrow Received(u_j(w))$ would involve significant scaffolding.

[9] One way to do that is to suppose that the network inputs include some signal that determines the rate and distribution of message delivery failure.

[10] TLA uses sequences of states, but these can be considered traces through a state machine.