Parameterized sequential functions for temporal properties

victor.yodaiken

October 2025

Abstract

Developers of "reactive" [4] software, like operating systems, still need to complete significant code development to have much idea about what performance and correctness problems or overlooked issues will force changes in the design or compromises in design goals. Temporal logic seemed to provide a way for developers to think about and validate these kinds of systems at a higher level, but progress has been slow at best. This paper shows that the kinds of properties that temporal logics promised to address, are among those that can be expressed in ordinary algebra with less notation and without the overhead of formal logic or axiomatic methods, using sequential functions[14] to define ordinary state machines.

1 Introduction

> *Everybody who has worked in formal logic will confirm that it is one of the technically most refractory parts of mathematics.* — Von Neumann[8]

Temporal logic[11, 10, 7, 3, 5, 6] is based on an intuitively appealing claim that being able to specify *when* a property must be true can be useful in specifications of computer systems. Examples include properties like: "*Always* no more than one process can be accessing a mutually exclusive resource", and "*Eventually* every scheduled process must start to run", and "a server cannot send an acknowledgment *Until* it has received the request."

This paper shows how to express similar and more detailed properties without formal logics or other axiomatic system using *sequential functions* [14]. A sequential function is a map from finite sequences of events to outputs so that $f(w)$ is intended to be the output of a discrete state system in the state reached by following event sequence w from the initial system state. As seen below, these functions can specify detailed operation of large scale state systems as well as abstract properties and the behavior and architecture of composite and concurrent systems.

Section 2 starts with a short introduction to sequential functions and a definition of temporal-logic-style operators on boolean valued sequential functions so that $(\mathit{Always}\, P)(w)$, $(\mathit{Eventually}\, P)(w)$ and so on are well-defined and can be seen to be at least similar to the corresponding temporal qualifiers (which differ among themselves in any case). Section 3 tackles a number of examples, including real-time and concurrent systems, and contrasts specifications that do and do not use the temporal-logic-like operators. Section 4 covers a real-time example used in a paper by Abadi and Lamport[1] and includes a detailed proof. The final section tries for some context.

The goal of this project is perhaps anachronistic: it is to provide human developers of operating systems and other complex "reactive" software [4] with a mathematical basis for specifying and thinking about designs. The reader can form their own opinion about how successful this effort has been.

2 Basic methods and Temporal-like operators

Each sequential function is associated with an *alphabet* – the set of discrete symbols representing events that can can produce a state change. Every state change is due to some event in the alphabet, although it is common practice to not completely list the alphabet as shown below. State systems are deterministic, so any finite sequence of events determines the state reached by following the sequence from the initial state. Event sequences in this paper are always finite.

If w is a event sequence and f is a sequential function, then $f(w)$ is a output in the state reached by following w from the initial state. Following the zero length sequence ϵ from the initial state leaves the system state unchanged so $f(\epsilon)$ is the output in the initial state. Appending an event a to a sequence, $w \cdot a$, drives the system to the next state so $f(w \cdot a)$ is the output in the state reached *after* event a drives the system from the state

determined by w.

If z is a finite sequence, then w `concat` z concatenates z to w on the right and $f(w$ `concat` $z)$ is the output in the state reached by following z from the state determined by w. Primitive recursion on sequences completely defines a sequential function:

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

Simple composition

$$f_2(w) = h(f_1(w))$$

modifies the output. Compositions of multiple sequential functions e.g.

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

define direct products of state systems (which do not interact). To interconnect state systems, sequence functions which translate from sequences of connected system events to component events can model communication and parallel or concurrent state change. See section 3.3.

Propositions are given as boolean valued sequential functions that have values in the set $\{0, 1\}$ with 1 for true and 0 for false. Boolean expressions like $f(w) < c$ will be treated as abbreviations for characteristic boolean functions like

$$P(w) = \begin{cases} 1 & \text{if } f(w) < c \\ 0 & \text{otherwise} \end{cases}$$

Sequential functions depend on a single event sequence argument, which is usually given as the first argument. Multiple arguments can mean that the output set X consists of maps (generally not sequential functions), so $f(w, x, y, z) = (f_1(w))(x, y, z)$. It may be that $F(w) = (f_1(w), f_2(w), \cdots f_n(w))$ and $f(w, i) = (f(w))_i = f_i(w)$.

More details on sequential functions can be found in appendix B and in the earlier paper [14] where it is shown that sequential functions are essentially just Moore-type state machines and products of these machines. The focus here is on intuition and applications not the underlying automata theory.

2.1 Temporal style operators

Concatenation of finite sequences, sequence prefixes, and sequence length are all well known operations, but definitions can be found in appendix A in case of doubt.

Definition 1

$$(\mathit{Always}\, P)(w) = \forall z \text{ where } P(w \,\texttt{concat}\, z) \text{ is defined, } P(w \,\texttt{concat}\, z)$$

$$\begin{aligned}(\mathit{Eventually}\, P)(w) = \;& \exists n \geq 0 \text{ so that} \\ & \forall z \text{ where } P(w \,\texttt{concat}\, z) \text{ is defined and } \mathit{length}(z) \geq n, \\ & P(w \,\texttt{concat}\, z') \text{ for some prefix } z' \text{ of } z\end{aligned}$$

$$(P \;\mathit{Until}\, Q)(w) = \forall z \text{ where } P(w \,\texttt{concat}\, z) \text{ is defined and } P(w \,\texttt{concat}\, z) = 0$$
$$Q(w \,\texttt{concat}\, z') \text{ for some prefix } z' \text{ of } z$$

$$\begin{aligned}(\mathit{Next}\, P)(w) = \;& \text{for every } a \text{ in the alphabet of } P \\ & \text{if } P(w \cdot a) \text{ is defined } P(w \cdot a)\end{aligned}$$

Note that if $P(w)$ then $(\mathit{Eventually}\, P)(w)$ with $n = 0$. ‘ The treatment of undefined states above is not necessarily the best for every application. It might also to be good to require that neither next nor eventually are trivially true, with at least one a so that $P(w \cdot a)$ is defined and one sufficiently long sequence z so that $P(w \,\texttt{concat}\, z)$ is defined.

2.1.1 Some Conventions

An expression $(\mathit{Eventually}\, P(w, x))$ has the meaning $(\mathit{Eventually}\, P_x)(w)$ where $P_x(w) = P(w)(x)$

Nested temporal style operators sometimes require a little care – like nested summations or integrals do. Consider $(\mathit{Always}(\mathit{Eventually}\, P))(w)$. Let $Q(w) = (\mathit{Eventually}\, P)(w)$. Then the original expression can be written $(\mathit{Always}\, Q)(w)$ which means that for all z so that $Q(w \,\texttt{concat}\, z)$ is defined, $Q(w \,\texttt{concat}\, z)$. Expanding Q the last expression means

$$(\mathit{Eventually}\, P)(w \,\texttt{concat}\, z)$$

which means that there is some n so that if u has length n or more, for some prefix u' of u,

$$P(w \,\texttt{concat}\, z \,\texttt{concat}\, u').$$

Suppose $\mathit{Sent}(w, i, m)$ is true iff network element i has sent message m, $\mathit{Received}(w, i, m)$ has the obvious meaning and $\mathit{Sending}(w, i, m)$ is true if i is currently sending the message. For these, suppose that $\mathit{Sent}(w, i, m) >$

$Sent(w \cdot a, i, m)$ is impossible so that a message never becomes unsent, and the same for *Received*. Then

$$\text{If } Sending(w, i, m) \text{ then } (Eventually\, Sent)(w, i, m) \tag{1}$$

would be a convenient property. If message m must be acknowledged with message m' and

$$\begin{array}{c} \text{If } Sent(w, i, m) > Received(w, i, m') \\ \text{then } (Eventually\, Sending)(w, i, m)) \end{array} \tag{2}$$

requires that element i keep sending the message until it gets the acknowledgment.

Alternatively, define $(Since\ P)(w)$ to be the count of events since P first became true.

$$(Since\ P)(\epsilon) = 0$$

$$(Since\ P)(w \cdot a) = \begin{cases} P(w) & \text{if } (Since\ P)(w) = 0 \\ 1 + (Since\ P)(w) & \text{otherwise} \end{cases}$$

Then for some $n \geq 0$ $(Since\ Sending)(w, i, m)) * (1 - Sent(w, i, m))) < n$ is equivalent to 1 above and for some $n_1 \geq 0$ $((SinceSent)(w, i, m)) * (1 - Received(w, i, m')) < n_1$ for some fixed n_1, is the same constraint as in 2 above. These constraints assume that there are no errors, which is not always a good assumption. It may be better to allow the network element to conclude that a client has failed or is unreachable if for some k $(Since\ Sent)(w, i, m) > k$ and $Received(w, i, m) = 0$. Section 3.4 shows how to count elapsed time instead of the number of events.

3 Examples

3.0.1 Simple examples

Some simple examples illustrate and will be useful in section 4. The common theme is to define state recursively. The functions "*tail*" and "*head* used below are defined in appendix A.

Specification 1 *Components.*

- *S is a **store** over values V if $S(w \cdot v) = v$ for every $v \in V$. It is a store over values V with initial value v_0 if $S(\epsilon) = v_0$. Otherwise the initial value is not specified.*

- *T is a **toggle** with initial value $b \in \{0, 1\}$ if $T(\epsilon) = b$ and $T(w \cdot a) = 1 - T(w)$.*

- *Q is a **transparent queue** over set V if $Q(\epsilon) = \epsilon$ and*

$$Q(w \cdot a) = \begin{cases} Q(w) \cdot v & \text{if } a = (enq, v) \\ tail(Q(w)) & \text{if } a = deq \text{ and } Q \neq \epsilon \\ Q(w) & \text{otherwise} \end{cases}$$

- *Q is a **bounded queue** over set V with length k if $Q(\epsilon) = \epsilon$ and:*

$$Q(w \cdot a) = \begin{cases} Q \cdot v & \text{if } a = (enq, v) \text{ and } Length(Q(w)) < k \\ tail(Q)v & \text{if } a = deq \text{ and } Q \neq \epsilon \\ Q(w) & \text{otherwise} \end{cases}$$

- *Q is a **closed queue** over set V with length k if for some bounded queue with length k, Q_k over V*

$$Q(w) = \begin{cases} \epsilon & \text{if } Q_k(w) = \epsilon;; \\ head(Q_k(w)) & \text{otherwise} \end{cases}$$

  Here, $\epsilon \notin V$ is assumed.

3.1 Process scheduling

Suppose $Status(w) \in \{free, ready, running, blocked\}$ is the scheduling status of a process within an operating system and the event alphabet is $A_{sched} = \{schedule, wake, preempt, run, block, free\}$. Scheduling status can be represented by a simple state graph as in figure 1 or with the following sequential

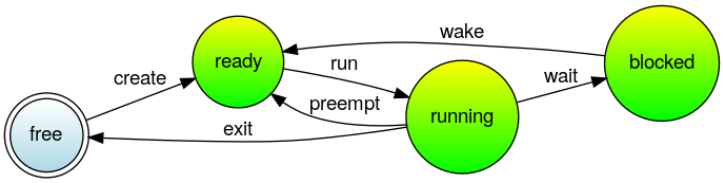

Figure 1: State graph of process scheduling status

function.

$$Status(\epsilon) = free$$

$$Status(w \cdot a) = \begin{cases} ready & \text{if } Status(w) = free \text{ and } a = schedule \\ & \text{or if } Status(w) = blocked \text{ and } a = wake \\ & \text{or if } Status(w) = running \text{ and } a = preempt \\ running & \text{if } Status(w) = ready \text{ and } a = run \\ blocked & \text{if } Status(w) = running \text{ and } a = block \\ free & \text{if } Status(w) = running \text{ and } a = exit \\ error & \text{otherwise} \end{cases}$$

The only difference between this state machine and the one defined by the state graph is the unrecoverable error state on an unexpected event. We could just forbid those events with $Status(w) \neq error$, but that constraint would then be on the wrong level because the schedule component does not control its inputs.

To track which core a process is run on replace the single *run* event with (run, c) for $c \in Cores$ where $Cores$ is a set of processor core identifiers. To track the *reason* a process is blocked, do the same thing with $(block, r)$ and a set of $Reasons$.

$$A_{sched} = \{schedule, wake, preempt, (run, c)\ c \in Cores, (block, r)\ r \in Reasons, free\}.$$

Then change the function definition.

$$Status(w{\cdot}a) = \begin{cases} ready & \text{if } Status(w) = free \text{ and } a = schedule \\ & \text{or if } Status(w) = blocked \text{ and } a = wake \\ & \text{or if } Status(w) = running \text{ and } a = preempt \\ (running, c) & \text{if } Status(w) = ready \text{ and } a = (run, c) \\ (blocked, r) & \text{if } Status(w) = running \text{ and } a = (block, r) \\ free & \text{if } Status(w) = running \text{ and } a = exit \\ error & \text{otherwise} \end{cases}$$

Even this change would make the graph in figure 1 impractical to draw if either the set of core names or the set of reasons had more than a very few elements.

The specification does not satisfy:

$$\text{If } Status(w) = ready \text{ then } (Eventually\, Status(w) = (running, c))$$

for some $c \in Cores$. The schedule component depends on the operating system for liveness as seen in section 3.3. The techniques of that section would also permit interconnection of other sequential functions that capture other process behavior. For example, $Req(w) \in Requests \cup \{null\}$ might capture the current request to the operating system, if any.

3.2 Clock pulse

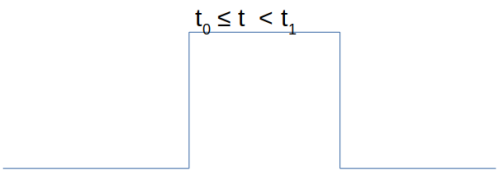

Figure 2: Specifying a clock signal

Suppose the system events are binary n-tuples that represent discrete time samples of input signals at a fixed frequency so $(a)_i$ is the signal on wire i carried by event (sample) a. Because the frequency is fixed, counting events tracks the passage of time. Say C is a circuit with m output signals

if $C(w) \in \{0,1\}^m$. Let $(C(w))_i$ be the i^{th} element of that output tuple. Suppose output k is supposed to be a clock signal with a cycle constrained by time parameters t_0, t_1 where $t_1 > t_0$ and the signal is supposed to stay stable *until* at least t_0 time units and then switch levels before t_1 time units have passed. See figure 2. What happens to the other output signals or what effects the input signals have on them is not specified, yet. Define $Stable(w, k, b)$ to count the time duration that output k has been stable at value $b \in \{0,1\}$.

$$Stable(\epsilon, k, b) = 0$$
$$Stable(w \cdot a, k, b) = ((C(w \cdot a))_k = b) * (1 + Stable(w, k, b))$$

Then the desired circuit property could be written:

$$\text{If } C(w)_k = b \text{ then } C(w)_k = b \; Until \; Stable(w, k, b) \geq t_0$$

and

$$Stable(w, k, b) < t_1$$

or we could replace the statement using *Until* with

$$\text{If } C(w)_k \neq C(w \cdot a)_k \text{ then } Stable(w, k, b) \geq t_0$$

3.3 Multiple processes

.

Suppose the schedule state systems from section 3.1 are components of an operating system or network system with its own event alphabet and there is set $ProcessIds$ of process identifiers. For $p \in ProcessIds$ a set of process identifiers, let:

$$Pstatus(w, p) = Status(u_p(w))$$

where u_p is a *connector* sequential function that translates sequences of the enclosing system into sequences over the component alphabet A_{sched}. When the system is in the state determined by w, schedule component p is in the state determined by $u_p(w)$. The constraint $Pstatus(w, p) \neq error$ requires the connector functions to not generate events that the schedulers don't expect.

Connector function definitions have two standard forms. The simpler case, where components advance by one or zero events on every event of the

enclosing system looks like this:

$$u_p(\epsilon) = \epsilon, \text{ and } u_p(w \cdot a) = u_p(w) \cdot g_p(w, a).$$

The equation $u_p(\epsilon) = \epsilon$ puts the components in their initial states when the operating system is in its initial state. The initial value can be any constant sequence over the component alphabet. The map g_p is called the tail function and $g_p(w, a) \in A_{sched} \cup \{\epsilon\}$ When the system advances by event a from the state determined by w, schedule component p advances by $g_p(w, a)$. To allow components to not advance at all on some system event, we can extend $\cdot$ so that $w \cdot \epsilon = w$ and require that $\epsilon \notin A_i$. The more general form allows components to advance by multiple steps on a single event for the enclosing state system (see appendix B).

There can be many different kinds of components in an interconnected system and figure 3 is a partial illustration of the operating system with components. For the *Req* sequential function of section 3.1 there might be connector maps v_p for each $p \in Processes$ and

$$g_p(w, a) = (block, r) \text{ only if } Req(v_p(w)) = r \text{ and}$$
$$\text{for some } c, \ Sched(u_p(w)) = (running, c)$$

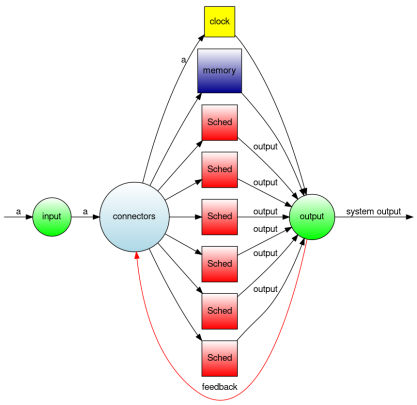

Figure 3: Multiple process state machines, interconnected

A requirement that no core is asked to run more than one process in any state constrains u_p.

$$\text{For each } c \in Cores \text{ there is at most one } p \in ProcessIds,$$
$$\text{so that } Pstatus(w, p) = (running, c)$$

This property could be made more concrete with:

$$\begin{array}{l}\text{If } g_p(w,a) = (run, c) \text{ then } \Sigma_p(Pstatus(w,p) = (running, c)) = 0 \\ \text{and } \Sigma_p(g_p(w,a) = (run, c)) = 1 \\ \text{and } Pstatus(w,p) = ready\end{array}$$

Now it makes sense to require scheduling be live.

$$\begin{array}{l}\text{If } Pstatus(w,p) = ready \text{ then} \\ (Eventually\, Pstatus(w,p) = (running, c)) \text{ for some } c\end{array}$$

This property could be a useful first approximation for example to show some network algorithm eventually gets to the right place. On the other hand, general purpose time-shared operating system generally cannot guarantee this property. We'd need at least an admissions policy to limit the number of active processes and additional controls on memory and other resources. A crude requirement to keep the number of active processes below the number of cores.

$$\Sigma_p(Pstatus(w,p) \neq free) \leq |Cores|$$

would be interesting only for very simple real-time systems. Attempting to specify how a general purpose operating system could guarantee liveness or at least identify the failure conditions that would break the guarantee, is a good problem. More realistically we could define a boolean sequence function to test whether timing specifications have been met, Then qualifying any propositions about the system with with this test limits those propositions to states where the timing specification has been met.

3.4 Real-time process constraints

The enclosing system can contain additional components along with processes. One component could be a real-time clock, which may not correspond to an actual device but could just be an ideal clock that somehow extracts time from events at the enclosing system level.

$$Clock(w \cdot a) > Clock(w) \tag{3}$$

In a deterministic event driven system, the passage of real-time must be represented by events since nothing else changes states but not all components need to change state as time advances. Which events cause time to pass can

differ by what physical system is being represented. For the circuit above described by figure 2, events are real-time samples of signal values so every event marks passage of one unit of time. In the operating system or network context, the passage of time might be marked by clock ticks from an oscillator, or clock interrupts, or some kind of input value like a time received from a GPS satellite or a vector of samples of signals on all input wires of the top level system during a discrete moment of time. It is also possible to take a real valued time variable t and just require that

$$Clock(w) \sim t \ . \tag{4}$$

That choice doesn't have to be made at this point. Let

$$Elapsed(w, a) = Clock(w \cdot a) - Clock(w). \tag{5}$$

Now it is possible to track how long a process has been runnable, waiting to run. Define $Delay(w, p)$ as follows:

$$Delay(\epsilon, p) = 0$$

$$Delay(w \cdot a, p) = \begin{cases} Delay(w,p) + Elapsed(w,a) & \text{if } Pstatus(w \cdot a, p) = ready \\ & \text{and } Pstatus(w,p) = ready \\ 0 & \text{if } Pstatus(w \cdot a, p) \neq ready \\ Delay(w,p) & \text{otherwise} \end{cases}$$

$Clock$ depends on w the sequence driving the operating system or network directly, while $Pstatus(w, p)$ depends on $u_p(w)$.

Define $RTSched(w, x)$ for $x > 0$ by $RTSched(\epsilon, x) = 1$ and

$$RTSched(w \cdot a) = \begin{cases} 0 & \text{if for some } p,\ Delay(w \cdot a, p) > x \\ RTSched(w,x) & \text{otherwise} \end{cases}$$

To require that every scheduled process must *eventually* run:

$$\text{for some } x_0 > 0, RTSched(w, x_0)$$

It is certainly possible to imagine the bound being state dependent, so

$$Delay(w, p) \leq Bound(w).$$

Bound might depend on how many processes are waiting to run and what resources they need and their priorities.

The constraint using the *Eventually* operator is not the same as any of these, because the eventually operator counts each event as one time unit and keeps the count invisible.

4 A Message queue

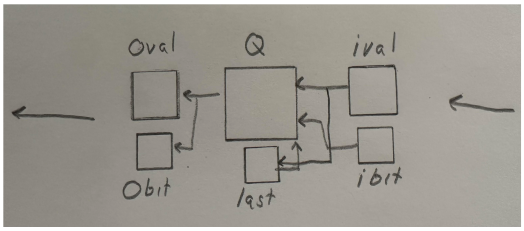

Figure 4: The queue from Abadi/Lamport

A paper by Abadi and Lamport [1] includes an example of a somewhat peculiar buffered queue. The queue is first specified in a way that makes it unreliable (it can lose input data) and then it is fixed by adding real-time constraints. I've tried to stay close to the approach in the Abadi and Lamport paper which they summarize as follows:

> *The interface consists of two pairs of "wires", each pair consisting of a* val *wire that holds a message and a boolean-valued bit wire. A message* m *is sent over a pair of wires by setting the* val *wire to* m *and complementing the* bit *wire. The receiver detects the presence of a new message by observing that the bit wire has changed value. Input to the queue arrives on the wire pair (*ival, ibit*), and output is sent on the wire pair (*oval, obit*). There is no acknowledgment protocol, so inputs are lost if they arrive faster than the queue processes them. (Because of the way* ibit *is used, inputs are lost in pairs.) The property guaranteed by this lossy queue is that the sequence of output messages is a subsequence of the sequence of input messages*[1]*. [Below] we add timing constraints to rule out the possibility of lost messages.* (page 1545)

The core of the pre-realtime Abadi and Lamport specification is in figure 5 but the reader here doesn't need to examine it to understand this specification.

The motivating example might be a A/D device or simple input device that has a single datum (message) input buffer, a program or device that is

[1]This is not correct. VY

responsible for collecting and queuing the messages, and a third component that dequeues messages and makes them available to a consumer in the output buffer. The queue is lossy because is is possible that messages that are stored to the input buffer can be overwritten by a second *input* event before they are moved into the internal queue.

From the outside, ignoring the components for the moment, suppose there is an alphabet A_{queue} which contains events including deq and $(input, m)$ for every message $m \in Msgs$ and other events which can be left for later. The output is a pair of the message at the head of the queue and a bit that alternates on each message.

$$(oval, obit) \Leftarrow \boxed{\mathrm{Q}} \Leftarrow enq, (input, m)$$

The output message is $nullm$ if the queue was empty on the last deq.

Here, the buffered queue is constructed from the store, toggle, and queue components from specification 1 above. There are three toggles and two buffers and one transparent queue. A $(input, m)$ event stores message m in the input buffer and toggles the input toggle. A deq event causes the internal queue to move the head of the queue to the output buffer and toggles the output toggle. An internal enq event copies the message in the input buffer to the internal queue and flips the $Last$ toggle bit. The components of the queue and connector functions are given below, with component names starting with a capital letter (e.g. $Last$) and connector functions all lower case: $last$.

Specification 2 *There is a set of messages Msgs. Inval and Outval are stores over Msgs with initial value nullm; Q is a transparent queue over Msgs; Inb, Last, and Outb are toggles with initial value* 0.
The connector maps are $inv, ov, ob, inb, last,$ *and* q*: inv for Inval, inb for Inb, ov for Outval, ob for Outb, last for Last, q for Q.*

The initial values of the connector functions are all ϵ.

$$inv(\epsilon) = ov(\epsilon) = inb(\epsilon) = last(\epsilon) = ob(\epsilon) = q(\epsilon) = \epsilon$$

The toggles all start out with output 0 per the specification. The message buffers are both set to $nullm$ initially. For example, since $inv(\epsilon) = \epsilon$ and $Invar$ is a store with initial value $nullm$, $Invar(inv(\epsilon)) = nullm$.

Each component advances only one or zero steps for every composite system event in A_{queue} since we let $w \cdot \epsilon = w$.

All connector maps have the form

$$u(\epsilon) = c,\ u(w \cdot a) = u(w) \cdot u^{\mathsf{T}}(w, a)$$

so specifying the recursive tail is most of what is needed. The convention that the tail of u is named u^{T} is just a naming convention, the $^{\mathsf{T}}$ doesn't have any other meaning.

An $(input, m)$ event produces an event stores m to the $Inval$ component and toggles the Inb toggle. Use the convention that $(E \neq E')$ has value one if the two expressions are not equal and 0 otherwise and then set

$$WaitEnq(w) = (Line(line(w)) \neq Inb(inb(w)))$$

The intent of the protocol with the toggles is that $WaitEnq(w) = 1$ when $Inval(iv(w))$ is an input message that has not yet been queued on the internal queue. The most complex connection map is q which controls events for the input queue and which is not completely specified but is constrained from enqueuing unless the $Last$ and Inb values differ. Incomplete specification takes the place of nondeterminism.

Specification 3 *The recursive "tails" of the connectors.*

$$inv^{\mathsf{T}}(w,a) = \begin{cases} a & \textit{if } a = (input, m) \textit{ for some } m \in Msgs \\ \epsilon & \textit{otherwise} \end{cases}$$

$$inb^{\mathsf{T}}(w,a) = inv^{\mathsf{T}}(w,a)$$

$$last^{\mathsf{T}}(w,a) = \begin{cases} enq & \textit{if } q^{\mathsf{T}}(w) = enq \\ \epsilon & \textit{otherwise} \end{cases}$$

$$q^{\mathsf{T}}(w,a) = \begin{cases} deq & \textit{if and only if } a = deq \\ (enq, m) & \textit{only if } a \neq deq \\ & \textit{and } m = Inval(in(w)) \\ & \textit{and } WaitEnq(w) \\ \epsilon & \textit{otherwise} \end{cases}$$

$$ov^{\mathsf{T}}(w,a) = \begin{cases} head(Q(q(w))) & \textit{if } a = deq \\ & \textit{and } Q(q(w)) \neq \epsilon \\ \epsilon & \textit{otherwise} \end{cases}$$

$$ob^{\mathsf{T}}(w,a) = ov^{\mathsf{T}}(w,a)$$

Finally:

$$Queue(w) = (Oval(ov(w)), Obit(ob(w)))$$

4.1 Real-time

To patch up the poor design of the protocol in this queue, Abadi and Lamport propose to put timing constraints on input and enq operations. The basic idea here is that when an *input* event happens, a timer starts and the matching *enq* must happen before t_0 seconds pass, but the next input event cannot happen until more than t_0 seconds have passed. It's pretty obvious that these timing constraints fix the problem with dropped messages, but see section 4.2.

Let's use the *Clock* described in equation 3 and 4 and *Elapsed* from equation 5 above.

$$(oval, obit) \Leftarrow \boxed{\text{Q}} \Leftarrow enq, (input, m)$$
$$time \Leftarrow \boxed{\text{Clock}} \Leftarrow A_{queue}$$

SinceI tracks the time that has passed since the last *input* event. In the initial state, it is set to t_0 so that the initial *input* can happen at any time without flagging an error.

$$SinceI(\epsilon) = t_0$$
$$SinceI(w \cdot a) = \begin{cases} 0 & \text{if } a = (input, m) \\ & \text{for some } m \in Msgs \\ SinceI(w) + Elapsed(w, a) & \text{otherwise} \end{cases}$$

There cannot be an *input* event without triggering a timing error *until* $SinceI(w) > t_0$. There must be an *enq* event *before* $SinceI(w) \geq t_0$.

Putting these together the timing requirement is:

$$Tspec(\epsilon) = 1$$
$$Tspec(w \cdot a) = \begin{cases} 0 & \text{if } WaitEnq(w \cdot a) \text{ and } SinceI(w \cdot a) \geq t_0 \\ & \text{or if } a = (input, m) \\ & \text{for some } m \in Msgs \\ & \text{and } SinceI \leq t_0 \\ Tspec(w) & \text{otherwise} \end{cases} \tag{6}$$

Once $Tspec(w) = 0$ it stays that way, because a single timing error is not recoverable for this queue.

The timing specification is not requiring that events follow the specification which could be done with $Always\, Tspec(w) = 1$. Instead the specification allows us to only consider states that are reached without violations of the specification.

For comparison, see page 1553 of [1] which is similar at least in intent.

4.2 Correctness

We want to prove that if $Tspec(w) = 1$ then the queue has not dropped or reordered or invented any queue elements. Abadi and Lamport characterize this property as something like, the sequence of input values is a prefix of

the sequence of output values. We're going to prove something stronger but first, let's define what those two sequences are.

Define: $Q_I(\epsilon) = \epsilon$ and

$$Q_I(w \cdot a) = \begin{cases} Q_I(w) \cdot m & \text{if } a = (input, m) \text{ for some } m \in Msgs \\ Q_I(w) & \text{otherwise} \end{cases}$$

This is the abstract queue of inputs. Define: $Q_O(\epsilon) = \epsilon$ and

$$Q_O(w \cdot a) = \begin{cases} Q_O(w) \cdot m & \text{if } q^{\mathsf{T}}(w, a) = (deq, m) \text{ for some } m \in Msgs \\ Q_O(w) & \text{otherwise} \end{cases}$$

This is the queue of messages that has passed through the device to the output buffer. The queues are similar to the *history variables* of Abadi and Lamport. The theorem (it's really too simple to be called a theorem) to prove is stronger than what Abadi and Lamport suggest.

Theorem 1 *Assuming* $Tspec(w)$

$$\textit{If } WaitEnq(w) = 0 \textit{ then } Q_I(w) = Q_O(w) \,\texttt{concat}\, Q(q(w))$$

$$\textit{If } WaitEnq(w) = 1 \textit{ then } Q_I(w) = Q_O(w) \,\texttt{concat}\, Q(q(w)) \cdot Inval(in(w))$$

The difference is that when $WaitEnq(w) = 1$ the value in the input buffer is the last item that has shown up on the input queue, otherwise the input buffer can be ignored. It's an immediate consequence of this theorem that $Q_O(w)$ is a prefix of $Q_I(w)$.

The proof of 1 is by induction on the event sequence. Initially, when $w = \epsilon$, all the queues are empty and $WaitEnq(\epsilon) = 0$ so there is nothing to prove. Suppose the theorem is true for w and consider $w \cdot a$. We assume $Tspec(w \cdot a)$ otherwise there is nothing to prove. There are four cases.

1. case:
$$WaitEnq(w) = 0 \text{ and } WaitEnq(w \cdot a) = 1$$
By the definition of $WaitEnq$, one of the two toggles has to change thanks to a. If $q^{\mathsf{T}}(w, a) = (deq, m)$ that would change the *Last* toggle but because $WaitEnq(w) = 0$ $q^{\mathsf{T}}(w, a) \neq (deq, m)$ for any m so it must be that $a = (input, m)$ for some m and Inb changes. By the inductive assumption and because $WaitEnq(w) = 0$ $Q_I(w) =$

$Q_O(w)$ `concat` $Q(q(w))$. Because $a = (input, m)$, $Q_I(w \cdot a) = Q_I(w) \cdot m$. By the definition of inv^{T} and because $a = (input, m)$, $Inval(inv(w \cdot a)) = m$. We know $a \neq deq$ so $q^{\mathsf{T}}(w, a) \neq deq$ – so $Q(q(w)) = Q(q(w \cdot a))$. This proves the first case.

2. case:
$$WaitEnq(w) = 0 \text{ and } WaitEnq(w \cdot a) = 0$$
In this case $a \neq (input, m)$ because otherwise $WaitEnq(w \cdot a) = 1$ unless the *Last* toggle flips too. But *Last* cannot change because that needs $q^{\mathsf{T}}(w, a) = (enq, m)$ and that cannot be true if $WaitEnq(w) = 0$. So $Q_I(w \cdot a) = Q_I(w))$. If $a = deq$ then if $Q(q(w)) = \epsilon$ nothing happens to Q_O or Q. If $Q(q(w)) \neq \epsilon$ and $a = deq$, then $q^{\mathsf{T}}(w, a) = deq$ so $Q(q(w \cdot a) = Tail(Q(w))$ and $Q_O(w \cdot a) = Q_O(w) \cdot Head(Q(q(w)))$ but
$$Q_O(w) \cdot Head(Q(q(w)) \,\texttt{concat}\, Tail(Q(q(w))$$
must be the same as
$$Q_O(w) \,\texttt{concat}\, Q(q(w)).$$
Otherwise $q^{\mathsf{T}}(w, a) = \epsilon$ That takes care of this case.

3. case:
$$WaitEnq(w) = 1 \text{ and } WaitEnq(w \cdot a) = 1$$
Because $WaitEnq$ remains true, $q^{\mathsf{T}}(w, a) \neq (enq, m)$ for any m so no new elements are added to $Q(q(w \cdot a))$. Because we assume $Tspec(w \cdot a)$, $WaitEnq(w) * SinceI(w) < t_0$ so a is not an input event so $inv^{\mathsf{T}}(w, a) = \epsilon$ and $Inval(inv(w \cdot a)) = Inval(inv(w))$. If $a \neq deq$, it must be that $q^{\mathsf{T}}(w, a) = \epsilon$ so $Q(q(w \cdot a)) = Q(q(w))$ when $a \neq deq$ and similarly $Q_O(w \cdot a) = Q_O(w)$ because $ov^{\mathsf{T}}(w, a) = \epsilon$. If $a = deq$ then than $Q(q(w \cdot a)) = Tail(Q(q(w)))$ and $Q_O(w \cdot a) = Q_O(w) \cdot Head(Q(q(w)))$ so the constructed queue is unchanged.

   For any other a nothing changes in the outputs of any components and we are done with this case.

4. case:
$$WaitEnq(w) = 1 \text{ and } WaitEnq(w \cdot a) = 0$$

Since $WaitEnq(w) = 1$

$$Q_I(w) = Q_O(w) \,\texttt{concat}\, Q(q(w)) \cdot Inv(inv(w).$$

Either $SinceI(w) < t_0$ or $TSpec(w) = 0$. We're assuming the second is false, so by the same reasoning $a \neq (input, m)$ (or else $Tspec(w{\cdot}a) = 0$). That means that $q(w, a) = (enq, Inv(inv(w)))$ to make $WaitEnq(w \cdot a) = 0$. As a result

$$Q(q(w \cdot a)) = Q(q(w)) \cdot Inv(inv(w)).$$

And the rest follows.

4.3 Some notes

The original specification depends on an unbounded queue, but in this case imposing a bound k on the queue could be accomplished by replacing the transparent queue with a bounded queue of length k which would make an enq operation when the queue is full just discard the message in the in buffer but would still toggle the value in *Last*. Given a timing specification for *deq* we could show that for some k the queue would be long enough. An interesting way to specify timing would be to require that deq operations happen every t_1 seconds at the latest, but that once a deq operation happened, other deq's would happen every t_2 for some $t_2 < t_1$ until the queue had been emptied.

5 Discussion

When Professor Krithivasan Ramamritham gave me temporal logic papers to read[12] at the start of my graduate studies in the 1980s, I had just finished working on a commerical distributed operating system[2] and was frustrated by how often the developers had found design errors only during test (after writing enormous quantities of code). Temporal logic seemed like a solution, but I was not unique in finding that it was not a completely satisfactory answer. Treatment of composition and concurrency, the inflexibility of the temporal quantifiers, the *ad hoc* nature of the formal semantics were among the issues that made it difficult to get a handle on the complexity of actual operating system design. Much of the methods used here and in [14] were

$$
\begin{array}{lcl}
Init_Q & \triangleq & \land\ ibit, obit \in \{\mathsf{true}, \mathsf{false}\} \\
 & & \land\ ival, oval \in \mathsf{Msg} \\
 & & \land\ last = ibit \\
 & & \land\ q = \langle\langle\rangle\rangle \\
Inp & \triangleq & \land\ ibit' = \neg ibit \\
 & & \land\ ival' \in \mathsf{Msg} \\
 & & \land\ (obit, oval, q, last)' = (obit, oval, q, last) \\
EnQ & \triangleq & \land\ last \neq ibit \\
 & & \land\ q' = q \circ \langle\langle ival \rangle\rangle \\
 & & \land\ last' = ibit \\
 & & \land\ (ibit, obit, ival, oval)' = (ibit, obit, ival, oval) \\
DeQ & \triangleq & \land\ q \neq \langle\langle\rangle\rangle \\
 & & \land\ oval' = Head(q) \\
 & & \land\ q' = Tail(q) \\
 & & \land\ obit' = \neg obit \\
 & & \land\ (ibit, ival, last)' = (ibit, ival, last) \\
\mathcal{N}_Q & \triangleq & Inp \lor EnQ \lor DeQ \\
v & \triangleq & (ibit, obit, ival, oval, q, last) \\
\Pi_Q & \triangleq & Init_Q \land \Box[\mathcal{N}_Q]_v \\
\Phi_Q & \triangleq & \exists q, last : \Pi_Q
\end{array}
$$

Fig. 2. The TLA specification of a lossy queue.

Figure 5: Abadi and Lamport's queue specification (untimed)

present in a 1991 paper [13], which included Moore machines, automata products for composition and concurrency and even a *modal* primitive recursive specification language. But that work was in the tradition of computational formal logic and this is not. When computer scientists started to try to develop mathematical methods of specifying programs, they often adopted methods from formal logic and other axiomatic methods perhaps because of the role of formal languages in programming languages and pattern matching and because of the influence of metamathematics on computability studies. Whatever the merits of that approach, discarding the apparatus of formal logic has allowed solution of a number of problems that blocked progress in

the earlier method and greatly simplified specifications.

The quote from Von Neumann that starts this paper is from a, not surprisingly, enormously perceptive paper that among other things previews algorithmic complexity theory. The full paragraph in which that quote is found has an interesting critique which I have tried to learn from as I returned to this work after a long absence.

> We are very far from possessing a theory of automata which deserves that name, that is, a properly mathematical-logical theory. There exists today a very elaborate system of formal logic, and, specifically, of logic as applied to mathematics. This is a discipline with many good sides, but also with certain serious weaknesses. This is not the occasion to enlarge upon the good sides, which I have certainly no intention to belittle. About the inadequacies, however, this may be said: Everybody who has worked in formal logic will confirm that it is one of the technically most refractory parts of mathematics. The reason for this is that it deals with rigid, all-or-none concepts, and has very little contact with the continuous concept of the real or of the complex number, that is, with mathematical analysis. Yet analysis is the technically most successful and best-elaborated part of mathematics. Thus formal logic is, by the nature of its approach, cut off from the best cultivated portions of mathematics, and forced onto the most difficult part of the mathematical terrain, into combinatorics.

References

[1] Martin Abadi and Leslie Lamport. "An old-fashioned recipe for real time". In: *ACM Trans. Program. Lang. Syst.* 16.5 (Sept. 1994), pp. 1543–1571. ISSN: 0164-0925. DOI: 10.1145/186025.186058. URL: https://doi.org/10.1145/186025.186058.

[2] Anita Borg, Jim Baumbach, and Sam Glazer. "A Message System Supporting Fault Tolerance". In: *SIGOPS Oper. Syst. Rev.* 17.5 (Oct. 1983), pp. 90–99. ISSN: 0163-5980. DOI: 10.1145/773379.806617. URL: https://doi.org/10.1145/773379.806617.

[3] 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*. 1983, pp. 117–119.

[4] D. Harel. “On the development of reactive systems.” In: *Logics and Models of Concurrent Systems*. Ed. by K. Apt. Springer-Verlag, 1985.

[5] Richard Alan Karp. “Proving Failure-Free Properties of Concurrent Systems Using Temporal Logic”. In: *ACM Trans. Program. Lang. Syst.* 6.2 (Apr. 1984), pp. 239–253. ISSN: 0164-0925. DOI: 10.1145/2993.357245. URL: https://doi.org/10.1145/2993.357245.

[6] L. Lamport. “The Temporal Logic of Actions”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (May 1994), pp. 872–923.

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

[8] John von Neumann. “The General and Logical Theory of Automata”. In: *Cerebral Mechanisms in Behavior*. Wiley, 1951, pp. 1–41.

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

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

[11] K. Ramamritham and R. M. Keller. “Specification of Synchronizing Processes”. In: *IEEE Trans. Softw. Eng.* 9.6 (Nov. 1983), pp. 722–733. ISSN: 0098-5589. DOI: 10.1109/TSE.1983.235435. URL: https://doi.org/10.1109/TSE.1983.235435.

[12] Krithivasan Ramamritham. “Correctness of A Distributed Transaction System”. In: *Information systems* 8.4 (1983), pp. 309–324.

[13] Victor Yodaiken. “Modal functions for concise definition of state machines and products”. In: *Information Processing Letters* 40.2 (1991), pp. 65–72. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(91)90011-6. URL: https://www.sciencedirect.com/science/article/pii/0020019091900116.

[14] Victor Yodaiken. "State Machines for Large Scale Computer Software and Systems". In: *Form. Asp. Comput.* 36.2 (June 2024). ISSN: 0934-5043. DOI: 10.1145/3633786. URL: https://doi.org/10.1145/3633786.

A Appendix: Sequence utilities

$$w \,\texttt{concat}\, \epsilon = w,\ w \,\texttt{concat}(z \cdot a) = (w \,\texttt{concat}\, z) \cdot a)$$

$$length(\epsilon) = 0,\ length(w \cdot a) = 1 + length(w)$$

$$prefixes(\epsilon) = \{\epsilon\},\ prefixes(w \cdot a) = prefixes(w) \cup \{w \cdot a\}$$

$$rappend(\epsilon, a) = a \cdot \epsilon, rappend(w \cdot b, a) = rappend(w, a) \cdot b)$$

$$Head(\epsilon) = \epsilon, Head(rappend(w, a)) = a$$

$$Tail(\epsilon) = \epsilon, Tail(rappend(w, a)) = w$$

B Appendix: Sequential functions

This is a short summary of results from [14]. The standard presentation of a Moore machine is $(A, X, S, s_0, \delta, \lambda)$ where $s_0 \in S$ is the start state, $\delta : S \times A \to S$ and $\lambda : S \to X$. Primitive recursive sequential functions are reformulations to make the state set implicit and the recursion explicit.

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

Such a map can be transformed into a conventional (possibly infinite state) Moore machine by letting the state set be the set $\{D(w) : w \in A^*\}$.

A sequential function is any map $f : A^* \to X$ where A is a set of events, A^* are the finite sequences over A and X is the set of outputs. Sequential functions define possibly infinite state Moore machines via the usual congruence on sequences.

The *primitive recursive* sequence functions are a proper subset with some useful properties including $f(w)$ must be defined if $f(w \cdot a)$ is defined. The *finite state* sequential functions are a proper subset of the primitive recursive sequential functions that correspond to finite state Moore machines. Specifications of real devices and programs tend to use primitive recursive and finite state primitive recursive sequential functions without any special attention

because properties of real systems are constructive and real systems are finite state machines anyway.

Primitive recursive sequential functions are constructed by a finite number of applications of the following rules.

1. **Constant**. If c is a constant $f(w) = c$ defines a sequential function.

2. **Constant map**. If g is a non-state dependent map, $f(w, x) = g(x)$ defines a sequential function.

3. **Primitive Recursion on Sequences**. If g is a non-state dependent map, c is a constant, ϵ is the empty sequence

$$f(\epsilon) = c,\ f(w \cdot a) = g(a, f(w)))$$

   defines a sequential function. If g has a finite image f is finite state.

4. **Simple Composition**. If f_1 is a sequential function, and $h(x)$ is an ordinary map

$$f(w) = h(f_1(w))$$

   defines a sequential function.

As with the arithmetic primitive recursive functions[9], many additional forms can be defined without changing the class of primitive recursive sequential functions. In particular multiple arguments $f(w, x_1, \ldots x_n)$ are fine, even with some arguments that are also state dependent $f(w) = (f(v_1(w), \ldots f_n(w)))$. An analog of arithmetic *simultaneous recursion* allows definition of composite system which correspond to automata products and specify concurrent/parallel systems of machines.

- **General Product**
  Given sequential functions $f_1, \ldots f_n$, each $f_i : A_i^* \to X_i$ and appropriate maps $g_i : A \times X \to A_i^*$ (not sequence dependent) where $X = X_1 \times \ldots X_n$,

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

$$u_i(\epsilon) = \epsilon,\ \text{and}\ u_i(w \cdot a) = u_i(w)\,\texttt{concat}\,g(a, i, v(w))$$

As shown in [14]: If each of the f_i is primitive recursive, then f is primitive recursive. If each f_i is finite state and A is finite then f is finite state.