A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol

J Strother Moore

Technical Report 68 1992

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

Abstract

In this paper we present a formal model of asynchronous communication as a function in the Boyer-Moore logic. The function transforms the signal stream generated by one processor into the signal stream consumed by an independently clocked processor. This transformation ‘‘blurs’’ edges and ‘‘dilates’’ time due to differences in the phases and rates of the two clocks and the communications delay. The model can be used quantitatively to derive concrete performance bounds on asynchronous communications at ISO protocol level 1 (physical level). We develop part of the reusable formal theory that permits the convenient application of the model. We use the theory to show that a biphase mark protocol can be used to send messages of arbitrary length between two asynchronous processors. We study two versions of the protocol, a conventional one which uses cells of size 32 cycles and an unconventional one which uses cells of size 18. Our proof of the former protocol requires the ratio of the clock rates of the two processors to be within 3% of unity. The unconventional biphase mark protocol permits the ratio to vary by 5%. At nominal clock rates of 20MHz, the unconventional protocol allows transmissions at a burst rate of slightly over 1MHz. These claims are formally stated in terms of our model of asynchrony; the proofs of the claims have been mechanically checked with the Boyer-Moore theorem prover, NQTHM. We conjecture that the protocol can be proved to work under our model for smaller cell sizes and more divergent clock rates but the proofs would be harder. Known inadequacies of our model include that (a) distortion due to the presence of an edge is limited to the time span of the cycle during which the edge was written, (b) both clocks are assumed to be linear functions of time (i.e., the rate of a given clock is unwavering) and (c) reading ‘‘on an edge’’ produces a nondeterministically defined value rather than an indeterminate value. We discuss these problems.

Keywords: hardware verification, fault tolerance, protocol verification, clock synchronization, Manchester format, FM format, automatic theorem proving, Boyer-Moore logic, ISO protocol level 1, performance modeling, microcommunications.

1. Introduction

In this paper we will (a) formalize the lowest-level communication between two independently clocked digital devices, (b) formalize the statement that, under certain conditions on the clock rates of the two processors, a biphase mark protocol permits the communication of arbitrarily long messages under our model of asynchronony, and (c) describe a mechanically checked formal proof that the statement is a theorem. Put less pedantically, we will exhibit a formal model of asynchronous communication and use it to prove that a commonly used protocol works.

We have tried to make this paper accessible both to hardware engineers, who are familiar with such terms as ‘‘asynchronous,’’ ‘‘clock rates’’ and ‘‘digital phase locking,’’ and to theorists, who are familiar with ‘‘formalize,’’ ‘‘theorem’’ and ‘‘proof.’’ Our attempt to bridge the gap between these two communities is largely found in the optional ‘‘boxes’’ scattered throughout the paper. There we try to explain possibly unfamiliar terms without detracting from what is otherwise a direct presentation of our formal model and the example of its use.

The biphase mark protocol—variously known as ‘‘Bi-ϕ-M,’’ ‘‘FM’’ or ‘‘single density’’ and sometimes called a ‘‘format’’ rather than a ‘‘protocol’’—is a convention for representing both a string of bits and clock edges in a square wave. Biphase mark is widely used in applications where data written by one device is read by another. For example, it is an industry standard for single density magnetic floppy disk recording. It is one of several protocols implemented by such commercially available microcontrollers as the Intel 82530 Serial Communications Controller [17] (where it is implemented with digital phase locking). A version of biphase mark, called ‘‘Manchester,’’ is used in the Ethernet [28] and is implemented with digital phase locking in the Intel 82C501AD Ethernet Serial Interface [17]. Biphase mark is also used in some optical communications and satellite telemetry applications [30]. There is no doubt that it works. But, as far as we have been able to determine, a rigorous analysis of its tolerance of asynchrony has not been done. This is a grey area because it is at the boundary between continuous physical phenonmenon (e.g., waves and interference) and discrete logical phenomenon (e.g., counting and algorithms).

Nevertheless, despite the apparent novelty of a rigorous analysis of a fundamental protocol, this paper is not really about the protocol. It is about a formal, logical model of asynchrony. We look at biphase mark only to illustrate how the model can be used.

Whether the assumptions in our model are valid is an engineering problem; indeed, accurately modeling the environment in which a device is expected to work may be the hardest problem the engineer faces. We offer no solution to that problem. In some sense there *is* no solution to that problem. It is up to the engineer to decide if a given model is accurate enough.

By expressing the model formally, one is forced to characterize precisely the requirements and assumptions. This done, one is then free to analyze them rigorously. In fact, we use mechanical aids that make the analysis both easier and less error prone.

In Figure 1 we illustrate the difficulty of interfacing two independently clocked devices (see box). The figure shows what might happen if one device sends the signal stream ‘‘**`tffftt`**’’[1] to an asynchronous receiver whose clock is half-again slower and initially almost one full cycle out of phase.

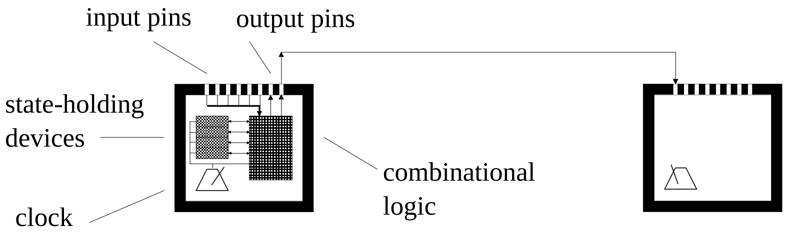
---

[1] In this paper we use **`t`** and **`f`** to denote the Boolean values of ‘‘truth’’ and ‘‘falsity.’’ These are also the values we use for ‘‘bits’’ (instead of **`1`** and **`0`**) and ‘‘signals’’ (instead of ‘‘high’’ and ‘‘low’’). Because timing diagrams are helpful in explaining our model, we adopt the convention that **`t`** is pictured ‘‘high’’ and **`f`** is pictured ‘‘low.’’

Microprocessors

A microprocessor is finite-state machine. Shapes and sizes vary but it is not inappropriate to imagine a rectangular piece of material about as large as a fingernail. On the outer edges of the rectangle are gold pins that allow electrical connections to other devices. We partition the pins into ‘‘input’’ and ‘‘output’’ pins (though in many devices some pins are both) with the understanding that the device is sensitive to the voltages on the former and sets the voltages on the latter. These voltages are called ‘‘signals.’’ When the voltage is above a certain threshhold it is called ‘‘high’’ and when it is below some other threshhold it is called ‘‘low.’’ Intermediate voltages are discussed below. Inside the rectangle are the memory devices, which store the state of the machine, and combinational logic, a network of wires and Boolean logic ‘‘gates,’’ for computing new states and output signals as a function of the old state and input signals. A metronome-like clock (usually a quartz crystal) ticks constantly during the operation of the machine. Typical clock rates are 20MHz, which means the clock ticks twenty million times a second. Each time the clock ticks the machine changes state. The state change is not instantaneous; it may take an appreciable portion of the cycle from one tick to the next for the new state to stabilize. Exactly when during the cycle the machine ‘‘reads’’ and ‘‘writes’’ its pins is entirely dependent upon the internal design of the machine.

An intermediate voltage on a pin may cause the device to behave contrary to this description. In particular, it may create a ‘‘metastable’’ state. Such a state may may appear to oscillate between different defined states or may spontaneously decay into a stable defined state independently of the clock and the mathematically understood state-transition function. Hardware designers strive to avoid the possibility that intermediate voltages appear on pins.



Now suppose we have two such processors. They are ‘‘asynchronous’’ with respect to each other because their clocks are independent. Suppose we connect an output pin of one to an input pin of the other. On every cycle of the first processor, some signal is written to the output pin and thus, after a suitable delay, appears at the input pin of the other processor. But because of the asynchrony, more than one signal may appear on the pin during a single cycle; the signal actually sensed and used to compute the next state of the receiving processor may be ill-defined or nondetermistically defined.

The problem of interfacing two such processors is a common one and usually occurs whenever a digital computer is connected to any other digital device (e.g., a modum, a disk drive, etc.).

Observe that in the ideal timing diagram, the signal falls from `t` to `f` on the writer’s second cycle. This is an idealization in two senses. First, the edge is not vertical or square, the signal changes continuously and may ‘‘ring’’ before stabilizing at its new level. Second, it does not happen immediately upon the clock tick that starts the second cycle. In fact, all that is promised by the ideal diagram is that the signal is stable and low by the end of the cycle. The funny looking ‘‘multivalued ramps’’ in the conservative model depicted in Figure 1 are intended to convey nothing more than that the signal is considered nondeterministically defined *throughout* the indicated cycles. We then impose upon that conservative diagram the receiver’s clock ticks. Consider the receiver’s first cycle. We do not know when during this cycle the receiver samples the signal (the time at which sampling occurs may be data dependent). But since the signal varies during the cycle, the exact time at which the receiver samples determines what is sensed. If it samples at

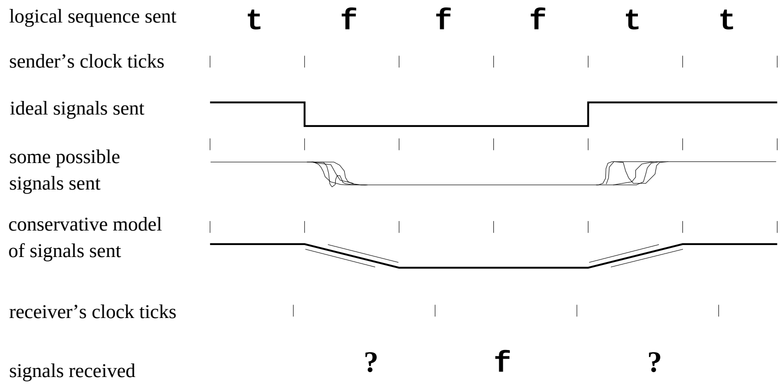


Figure 1: How Asynchrony Mangles Signals

the ''wrong'' time it could even read an indeterminate signal that could induce a metastable state. Things are simpler during the receiver's second cycle; the signal is constant at `f` for the duration of the cycle and hence we are assured that it reads an `f`.

The problem of metastability caused by reading on an edge cannot be solved perfectly by digital logic alone. By cascading the asynchronous signal through several state-holding devices on successive clock ticks of the receiver one can increase the probability that the signal stabilizes before it enters into the determination of the next state. Such cascades are called ''synchronizers'' but there is some degree of wishful thinking here since there remains a non-zero probability that the metastable state persists [22]. One can also build devices with hysteresis, e.g., Schmitt triggers, that require well-defined input before changing their output. Such devices can be used to sharpen an edge, but since these devices essentially just narrow the band of indeterminacy, there is still some chance that metastability will slip in. In summary, metastability is an engineering problem that apparently has no perfect solution. We do not attempt to model it. Our model assumes that ''reading on an edge'' nondeterministically produces a `t` or an `f`.[2] It is up to the engineer to arrange that some well-defined signal is read on each cycle.

This however does not solve the communications problem. Nondeterministically replacing the question marks in Figure 1 by `t`s and `f`s does not enable the recovery of the original signal stream. Even an accurate analysis of which read cycles produce nondeterministic signals or how many such cycles there are requires careful consideration of the two clock rates and their phase displacement. For example, as illustrated in Figure 2, if the rates are nearly identical (the usual case) and the receiver's cycle is the shorter, then, depending on the initial phase displacement (which can be arbitrary for two physically independent clocks), an edge in the arriving signals can affect two or sometimes three successive read cycles. Nondeterministically replacing the question marks by `t`s and `f`s has the effect of blurring or shifting the edges in the signal. Differences in the clock rates of the two processors may stretch or shrink the apparent duration of the signal.

Communications protocols have been developed to deal with these problems. To avoid the first problem,

---

[2] It is possible to model indeterminate signals logically. Three- and even four-valued logics are common in hardware description languages. We have mechanically proved that in one such logic it is impossible to build even a simple asynchronous edge detector with perfect reliability. The NQTHM transcript is available upon request.

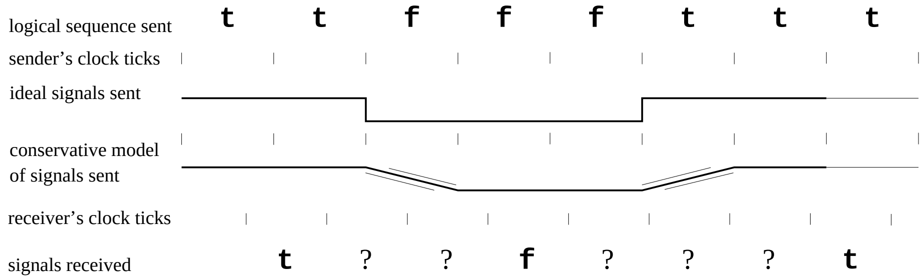


Figure 2: One Edge Can Influence Several Read Cycles

the asynchronous sender generally encodes its message as a waveform with a relatively long wavelength compared to the cycle time of the receiver, giving the receiver plenty of time to sample the signal away from the edges. To overcome the second problem, the biphase mark protocol encodes the message with ''frequency modulation'' of the long wavelength ''carrier.'' This allows the receiver to ''phase lock'' onto the artificially slower clock of the sender.

In the biphase mark protocol (see Figure 3), each bit of message is encoded in a ''cell'' which is logically divided into what we call a ''mark subcell''[3] and a ''code subcell.'' During the mark subcell, the signal is held at the negation of its value at the end of the previous cell, providing an edge in the signal train which marks the beginning of the new cell. During the code subcell, the signal either returns to its previous value or does not, depending on whether the cell encodes a **t** or an **f**. The receiver is generally waiting for the edge that marks the arrival of a cell. Upon detecting the edge, the receiver counts off a fixed number of cycles, here called the ''sampling distance,'' and samples the signal there. The sampling distance is determined so as to make the receiver sample in the middle of the code subcell. If the sample is the same as the mark, an **f** was sent; otherwise a **t** was sent. The receiver then resumes waiting for the next edge, thus ''phase locking'' onto the sender's clock.

Of course, asynchrony may blur or shift the edges of the code subcell, but if the code subcell is sufficiently long, some region of it (away from the edges) will be well-defined. We call this region the ''sweet spot.'' The receiver should always sample from the sweet spot. What might prevent this? A plausible scenario is that the receiver is late detecting the mark because of nondeterminism and then waits too long before sampling because its clock is slower than the sender's.

This scenario should make it clear that the extent to which the protocol relies upon the near agreement of the two clock rates is dependent upon how far the sweet spot is from the mark. It is while measuring out this time interval (while creating the cell in the sender or waiting to sample in the receiver) that the protocol implicitly assumes the two processors cycle at the same rate. If two clocks are used to measure out some absolute time interval, and the two clock's rates are fixed but slighly different, their discrepancy in the measurement is linearly proportional to the length of the interval measured. Thus, the closer the sweet spot

[3]The word ''mark'' in ''biphase mark'' comes from the ''Automatic Recorder'' of 19th century telegraphy where the line idle state produced a mark on a rotating drum and the arrival of a pulse lifted the stylus to produce a space [8]. The names MARK and SPACE were adopted for logical 1 and logical 0 respectively. However, except in the name ''biphase mark,'' our use of the word ''mark'' is intended in its nontechnical sense, i.e., ''a conspicuous object serving as a guide for travelers'' [24]. Thus we speak of the ''mark subcell,'' so named because it indicates the beginning of the cell, and of ''detecting the mark.''

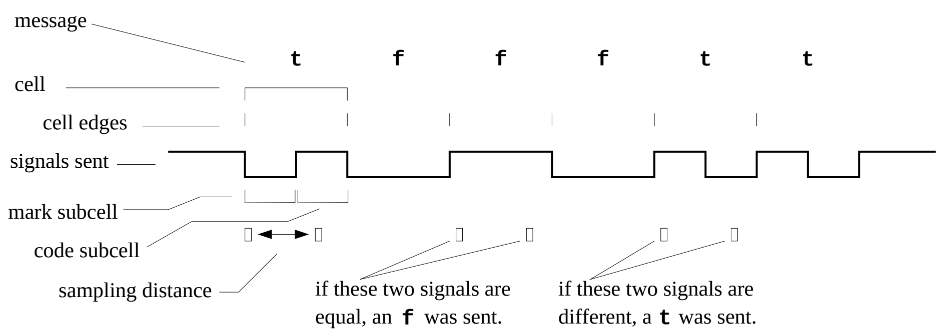


Figure 3: Biphase Mark Terminology

is to the mark, the more tolerant the protocol is to different clock rates.

To analyze the behavior of the protocol in the face of asynchrony we must specify the cell size, subcell sizes, and sampling distance. We study a conventional choice and an unconventional one. The conventional choice is cell size 32, equally divided into two 16-cycle subcells, sampled on the 23rd cycle after mark detection. The unconventional choice is cell size 18, divided into a 5-cycle mark and a 13-cycle code subcell, sampled on the 10th cycle after mark detection. The unconventional choice permits a faster bit rate (since fewer cycles are spent on each bit) and tolerates more divergent clock rates (since the time during which the clocks must ‘‘stay together’’ is smaller). Do they work?

In this paper we formally (see box) define a model of asynchrony and we formally state and prove the theorem that, under the model, the 18-cycle/bit biphase mark protocol properly recovers the message sent, provided the ratio of the two clock rates is between 0.95 and 1.05. According to [29] typical clocks are incorrect by less than 15×10^{-6} seconds per second and the ratio of the rates of two such clocks are well within our bounds. We have proved that the conventional choice of cell size also works, provided the ratio of the clock rates is within 3% of unity, and we briefly indicate how the proof differs from the proof of the 18-cycle version.

2. Logical Foundations

We use the NQTHM ‘‘computational logic’’ described in [6].

Truth values, bits, and signals will all be represented by the logical objects **`t`** and **`f`** which are distinct constants. We call these two objects the ‘‘Booleans’’ and we define a predicate, **`boolp`**, which recognizes just them.

Definition.
```
(boolp x)
  =
(or (equal x t) (equal x f))
```

Observe that, as in Lisp, we write function applications with the parentheses ‘‘on the outside.’’ Thus, we write **`(boolp x)`** to mean the value of the function **`boolp`** applied to **`x`**, i.e., **`boolp`**(**`x`**). As can be seen

Formalization

What do we mean when we say we define the model or state the theorem ‘‘formally?’’ We mean we exhibit a formula that purportedly captures the idea. Because we are interested in mathematical proof, we write our formulas in the language of a particular mathematical logic. A logic provides a language, some axioms (formulas assumed to be true), and some rules of inference (truth preserving operations on formulas). To prove a theorem is to derive that formula from the axioms using the rules of inference.

The logic we use is called the ‘‘NQTHM’’ or ‘‘Boyer-Moore’’ logic. Its language resembles Pure Lisp; its axioms define the primitive functions for if-then-else, equality, and list and number processing; its rules of inference include such familiar ones as ‘‘substitution of equals for equals,’’ ‘‘every instance of a theorem or axiom is a theorem’’ and mathematical induction. We will explain the logic as we go.

But how can we write a formula that says the biphase mark protocol works in the face of asynchrony? Because the NQTHM logic is essentially just a programming language without side-effects, the whole formalization problem can be recast as a programming problem:

Challenge: Write a Pure Lisp program (together with its subroutines) with the property that if the program returns **`t`** on all possible inputs then you will believe that the biphase mark protocol works.

Ah! This is a straightforward programming problem. The solution is to write a ‘‘simulator’’ for the system being modeled. That is, we will develop a Pure Lisp program that takes among its inputs a message to be tested, the precise clock rates of the two processors, and their initial phase displacement and delay, and simulates the encoding, sending, receiving, and decoding of the message. The ‘‘simulator’’ will return **`t`** if the message is recovered and **`f`** otherwise.

But how can such a simulator be run on all possible inputs? Clearly it cannot be. That is where we use proof. Since the simulator is just a collection of mathematically defined functions we can use substitution, instantiation, and induction to show that the simulator always returns **`t`**.

by an inspection of the definition, **`(boolp x)`** is **`t`** if and only if **`x`** is **`t`** or **`x`** is **`f`**.

The NQTHM logic imposes restrictions on equations purporting to be ‘‘definitions.’’ These restrictions insure that one and only one mathematical function satisfies the equation. Because of this assurance, we can add such admissible definitions to the logic without rendering the logic inconsistent. The reader should see [5, 6] for details. In this presentation we do not further concern ourselves with the admissibility of our definitions.

We define the operations of ‘‘negation’’ and ‘‘exclusive-or’’ as follows.

Definitions.
```
(b-not x) = (if x f t)

(b-xor x y)
   =
(if x
    (if y f t)
    (if y t f))
```

Thus, **`(b-not t)`** is **`f`** and **`(b-xor t f)`** is **`t`**.

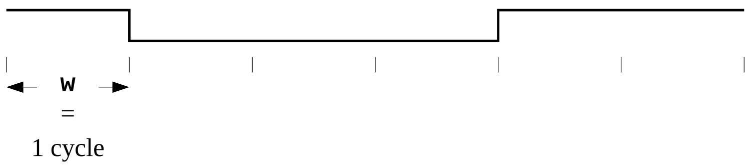
Fundamental to our formalization is the notion of a ‘‘bit vector’’ or a ‘‘finite sequence of Booleans.’’ We use lists (see box) to represent such objects. The following function recognizes bit vectors.

Definition.
```
(bvp x)
    =
(if (listp x)
    (and (boolp (car x))
         (bvp (cdr x)))
    (equal x nil))
```

That is, `(bvp x)` is defined by cases. If `x` is a `listp` object, then its first element, `(car x)`, must be Boolean and the rest of its elements, `(cdr x)`, must recursively satisfy `bvp`. On the other hand, if `x` is not a `listp` object, it must be `nil`. An example bit vector is `(list t t f f)`. That is, `(bvp (list t t f f))` evaluates to `t`.

We shall use `(len x)` to denote the length of the list `x`, `(app x y)` to concatenate the lists `x` and `y`, `(nth n x)` to fetch the $\mathbf{n}^{th}$ element of the list `x` (where `(car x)` is the $\mathbf{0}^{th}$ element), `(cdrn n x)` to `cdr` the list `x` `n` times, and `(listn n x)` to make a list of `n` repetitions of the object `x`. We omit the definitions of these simple functions.

We will use lists of Booleans (bit vectors) to represent streams of signals or ‘‘timing diagrams.’’ For example,



will be represented by the list `(list t f f f t t)` together with the fact that the length of a cycle is `w`. An alternative way of writing the same list is `(cons t (app (listn 3 f) (listn 2 t)))`.

3. The Model of Asynchrony

Consider two independently clocked processors, which we call the ‘‘writer’’ and the ‘‘reader.’’ The output pin of the former is connected by a wire to the input pin of the latter and this constitutes the only communication between them. Imagine that on successive cycles the writer is specified to set its output pin to the successive signals in some bit vector called the ‘‘writer’s view.’’ We wish to define a function, `async`, which will map the writer’s view into the sequence actually read by the reader, which we call the ‘‘reader’s view.’’

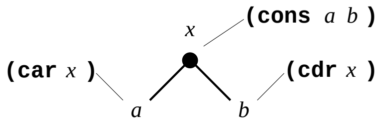
More precisely, we map the writer’s view into any one of the possible reader’s views, since there is an element of nondeterminancy here. One parameter of the model, called the ‘‘oracle,’’ specifies how each nondeterministic choice is to be made on a given application of the model; by varying this parameter one can obtain all possible views by the reader.

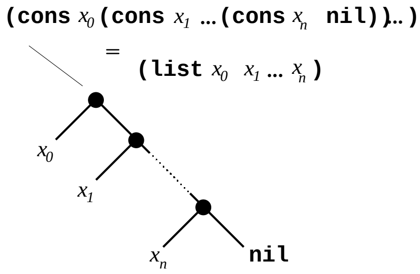
Our model is based on three assumptions.

- The distortion in the signal due to the presence of an edge is limited to the time-span of the cycle during which the edge was written. For example, we ignore intersymbol interference [28].
- The clocks of both processors are linear functions of real time, e.g., the ticks of a given clock are equally spaced events in real time. We ignore clock jitter.

List Processing

In the logic, lists are binary trees. Binary trees are ordered pairs constructed by the function **`cons`** from any two objects. The functions **`car`** and **`cdr`** return the two objects. The function **`listp`** recognizes just the objects produced by **`cons`**. That is, **`(listp x)`** is **`t`** or **`f`** depending on whether **`x`** is an ordered pair. We use **`(nlistp x)`** as an abbreviation for ‘‘non-**`listp`**.’’





Example Axioms

```
(car (cons a b)) = a
(cdr (cons a b)) = b
(listp (cons a b)) = t
(listp nil) = f
```

We frequently define recursive functions on lists. For example, the ‘‘length’’ of a list **`x`**, written **`(len x)`**, is defined

Definition.

```
(len x)
    =
(if (listp x)
    (add1 (len (cdr x)))
    0).
```

Such definitions are usually read as ‘‘by cases.’’ ‘‘If **`x`** is a **`listp`**, its length is obtained by adding one to the length of its **`cdr`**; if **`x`** is not a **`listp`**, its length is **`0`**.’’ Thus, **`(len (list t t f f))`** is **`4`**.

Given two lists, we can ‘‘concatenate’’ (or ‘‘append’’) them using the function **`app`**,

Definition.

```
(app x y)
    =
(if (listp x)
    (cons (car x) (app (cdr x) y))
    y).
```

We might paraphrase this as ‘‘To append a nonempty **`x`** to **`y`**, **`cons`** the first element (**`car`**) of **`x`** to the result of appending the rest (**`cdr`**) of **`x`** to **`y`**. To append an empty **`x`** to **`y`**, return **`y`**.’’ Thus,

```
(app (list 1 2) (list t f f))
    =
(cons 1 (app (list 2) (list t f f)))
    =
(cons 1 (cons 2 (app nil (list t f f))))
    =
(cons 1 (cons 2 (list t f f)))
    =
(list 1 2 t f f).
```

- Reading on an edge produces nondeterministically defined signal values, not indeterminate values.

Our model of asynchronous communication has three passes, one implementing each of the assumptions above. In Figure 4 we illustrate the passes. In pass 1, we identify those cycles in which the signal is

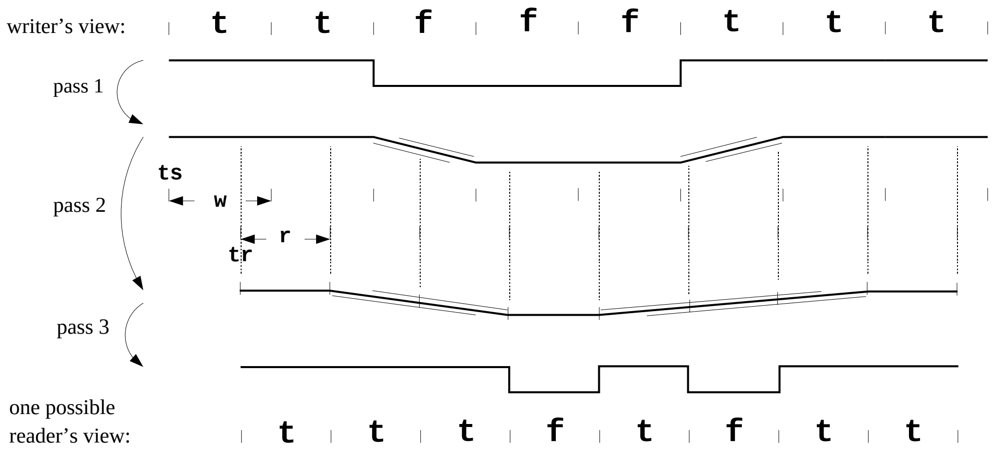


Figure 4: The Three Passes in the Model

undetermined due to the non-zero switching times on the writer. This is indicated in the graph in Figure 4 by the multivalued ramps on two of the write cycles. Pass 2 combines the pass 1 output with certain timing information (the cycle times, **`w`** and **`r`**, of the two processors and (roughly) their phase displacement, **`tr`**−**`ts`**) to produce the signal on the pin during each read cycle (up to nondeterminacy). Pass 2 is the key to the model and operates by reconciling all the signals on the pin during each read cycle. Pass 2 generally smears the nondeterminacy over any read cycles which overlap with it. Pass 2 may lengthen or shorten the length of the signal stream but does not change its basic shape. Pass 3 eliminates the nondeterminacy by using the oracle to choose arbitrary values for undetermined signals.

It should be noted that our model puts no constraints on the relationship between the writer's cycle time and the reader's. That is, one can apply this model to communication between two processors whose clocks run at wildly different rates. For example, if the reader runs ten times as fast as the writer, it will see roughly ten times more signals. The model is somewhat pathological if either processor runs infinitely fast (i.e., has a cycle time of 0). We do not constrain the relationship between the clocks until we begin to apply the model to prove that a certain protocol works.

We now back up and give a more detailed physical and formal explanation.

3.1 Pass 1

Consider the writer. On every cycle the writer sets the output pin to some value. If that value is the same as the previous value of the pin, then the signal on the pin remains stable at that value for the entire cycle. On the other hand, if the new value is different, then we assume the value on the pin is undetermined for the duration of that cycle. This accounts for our lack of knowledge about when during the cycle the voltage on the pin begins to change, how the voltage varies, and how long it takes it to become stable. Pass 1 in the model thus introduces ‘‘multivalued ramps’’ for the duration of every cycle during which the signal changes. The ramps in our diagrams are formally represented by the ''signal'' **'q**, which is just a token that will eventually be replaced nondeterministically. There is no need to distinguish ''downward'' ramps from ''upward'' ones since they both mean the signal is indeterminate for the entire cycle.

The function formalizing pass 1 is called **smooth** and it takes the previous signal seen, **x**, and a sequence of signals, **lst**.

Definition.

```
(smooth x lst)
   =
(if (nlistp lst)
    nil
    (if (b-xor x (car lst))
        (cons 'q
              (smooth (car lst) (cdr lst)))
        (cons (car lst)
              (smooth (car lst) (cdr lst)))))
```

Observe that **smooth** copies **lst**, changing to **'q** any signal that is different from the previous one, **x**. In Figure 4, pass 1 is computed by **(smooth t (list t t f f f t t t))**, which replaces the underscored signals by **'q**s.

3.2 Pass 2

Now, let **lst** be the output of pass 1. In pass 2 we simulate the arrival of these signals at the input pin of the reader, consider the reader's cycles, and compute the signals read (up to nondeterminacy). Suppose the first signal, **(car lst)**, arrives at the input pin at time **ts**.[4] All successive signals arrive at intervals of **w**, where **w** is the cycle time of the writer. Let **tr** be the time at which the reader's clock first ticks at or after **ts**. Without loss of generality we assume **ts** ≤ **tr** < **ts**+**w** because if **tr** ≥ **ts**+**w** then the first signal of **lst** is simply irrelevant since it does not persist into the reader's first cycle. Finally, suppose the reader's cycle time is **r**. Given these parameters we can compute the entire list of signals read (up to nondeterminancy). We call the function formalizing pass 2 **warp** and define it below.

Definition.

```
(warp lst ts tr w r)
   =
(if (or (zerop r)
        (endp lst ts (plus tr r) w))
    nil
    (cons (sig lst ts (plus tr r) w)
          (warp (lst+ lst ts (plus tr r) w)
                (ts+ lst ts (plus tr r) w)
                (plus tr r)
                w r)))
```

The term **(plus tr r)**, above, is the sum of **tr** and **r** and is the time at which the reader's clock next ticks. The definition may be read as follows: If **r** is zero[5] or else if **lst** does not have enough elements in it to determine the next signal read, return the empty list **nil**. The second condition is checked by **endp** which we discuss below. If **r** is nonzero and there are enough elements in **lst** to determine the next signal

[4]More precisely, consider that tick of the writer's clock that began the write cycle during which the first signal was written. Let ω be the time at which that tick occurred. Let δ be the delay along the wire connecting the writer to the reader. Then **ts** is ω+δ. We assume δ is constant.

[5]Omitting the **(zerop r)** test produces an inadmissible definition because the recursion described does not terminate.

read, we use **sig** (described below) to compute the signal read during the current cycle, we use **warp** recursively to obtain the list of signals read on successive cycles and then we **cons** together the two results to produce the list of all the signals read.

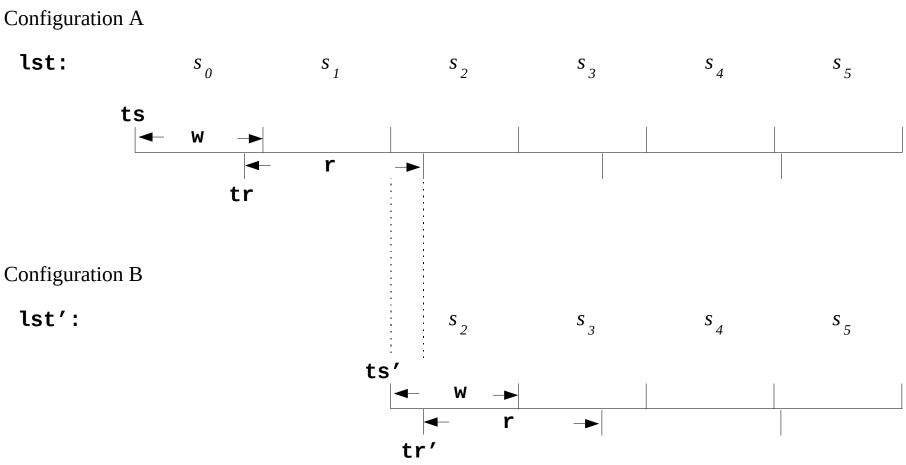


Figure 5: The Recursion in **warp**

We explain further by referring to Figure 5. Configuration A of the figure depicts the formal parameters of **warp** upon entry to **(warp lst ts tr w r)**. Note that **lst** contains six signals, s_0, ..., s_5 and that s_0 arrives at time **ts** and persists for time **w**. The first tick of the reader's clock is at time **tr** and starts a cycle that persists for time **r**. By observing the diagram in Configuration A we see that the signals s_0, s_1 and s_2 impinge upon the pin during this read cycle. If they are all equal, say, to s_0, then s_0 will be the signal read on this cycle. But if any two are different, the signal read is nondeterministic (i.e., **'q**). This is the computation made by **(sig lst ts (plus tr r) w)**.

Configuration B of Figure 5 shows the parameters passed to the recursive call of **warp** from Configuration A. The call in question is

```
(warp (lst+ lst ts (plus tr r) w)
      (ts+ lst ts (plus tr r) w)
      (plus tr r)
      w r)
```

The easiest argument term to understand is **(plus tr r)**, passed as the new value of **tr**. That is the time of the next tick of the reader's clock and is shown as **tr'** in Configuration B. The faint dotted line is meant to indicate that **tr'** is **tr**+**r** from Configuration A. **Lst'** is the new value of **lst**. Note that (in this case) the first two signals have been removed from **lst**. That is because they were used in the **sig** computation for the current cycle and do not affect the **sig** computation at the next cycle. Note that s_2, which was used by the **sig** computation, is still in **lst'** because it persists into the next cycle. **Lst'**, which is always some **cdr** of **lst**, is computed by the function **lst+** in the recursive call of **warp**. The time at which the new first signal arrives, **ts'**, is computed by the function **ts+**.

The four functions **endp**, **sig**, **lst+**, and **ts+** are all very similar in that they scan **lst**, knowing that the first signal arrives at time **ts** and that subsequent ones arrive at intervals of **w**, and look for the first signal that persists into the next cycle, i.e., the one that starts at **(plus tr r)**. The function **endp** returns **t** if **lst** is exhausted before the desired signal is reached. **Sig** reconciles all the signals it reaches, using the auxiliary function **reconcile-signals**. **Lst+** returns the **cdr** of **lst** starting with the desired signal. **Ts+** returns the arrival time of the desired signal. The definitions are shown below.

Definition.
```
(endp lst ts nxtr w)
    =
(if (nlistp lst)
    t
    (if (lessp (plus ts w) nxtr)
        (endp (cdr lst) (plus ts w)
              nxtr w)
        f))
```

Definition.
```
(sig lst ts nxtr w)
    =
(if (nlistp lst)
    'q
    (if (lessp (plus ts w) nxtr)
        (reconcile-signals
         (car lst)
         (sig (cdr lst) (plus ts w)
              nxtr w))
        (car lst)))
```

Definition.
```
(reconcile-signals a b)
    =
(if (equal a b) a 'q)
```

Definition.
```
(lst+ lst ts nxtr w)
    =
(if (nlistp lst)
    lst
    (if (lessp nxtr (plus ts w))
        lst
        (lst+ (cdr lst) (plus ts w)
              nxtr w)))
```

Definition.
```
(ts+ lst ts nxtr w)
    =
(if (nlistp lst)
    ts
    (if (lessp nxtr (plus ts w))
        ts
        (ts+ (cdr lst) (plus ts w)
             nxtr w)))
```

Readers familiar with NQTHM will have noticed that the arithmetic primitives used in **warp** treat their arguments as natural numbers. That is, **ts**, **tr**, **w**, and **r** in this model are nonnegative integers. Since time appears continuous, the reals or the rationals seem more appealing domains for these parameters. However, the NQTHM logic does not support the reals. The rationals have been defined within the logic and they were used when the model was first being formalized. However, the proof we will describe is primarily concerned with counting cycles. We found that the proof was complicated by the mix of (formal) natural arithmetic and (formal) rational arithmetic. We decided to simplify matters by adopting natural arithmetic entirely. It should be stressed that this is primarily a technical problem with the NQTHM mechanization and its heuristics.

Inspection of the model will reveal that our use of natural arithmetic does not limit the applicability of the model. In particular, if **ts**, **tr**, **w**, and **r** are given as rational numbers, one could convert them to four naturals over a common denominator and then do all the arithmetic on the numerators only, using natural arithmetic. This observation relies on the fact that the model only iteratively sums and compares these quantities. But $\frac{a}{d}+\frac{b}{d} = \frac{a+b}{d}$, where the first ‘‘+’’ is that for rational arithmetic and the second is that for natural arithmetic. A similar theorem holds for the ‘‘less than’’ relationships in the two systems.

An illustration of **warp** was presented in pass 2 of Figure 4. In that example, the input list was the output of pass 1, **(list t t 'q f f 'q t t)**, **ts** was **0**, **tr** was **75**, **w** was **100**, and **r** was **87**. The output of **warp** was **(list t 'q 'q f 'q 'q 'q t)**. We used grossly mismatched **w** and **r** merely so that it was easy to see that read cycle 5 (counting from 0) fell entirely within write cycle 5. Exactly identical signal output can be obtained with more realistically matched clocks. For example, let us measure time in tenths of picoseconds, e.g., units of 10^{-13} seconds. If the writer has a perfect 20MHz clock then **w** is 500,000. Suppose the reader is nominally 20MHz but ticks faster so that in twenty million ticks it counts off .999996 seconds. That is, **r** is 499,998 and the clock is gaining roughly 4×10^{-6} seconds per second, which is consistent with the clocks reported in [29]. Then if the first signal in the output of pass 1 reaches the reader 11×10^{-13} seconds before the reader's clock ticks, the output is as described in pass 2 of Figure 4. I.e., **(warp (list t t 'q f f 'q t t) 0 11 500000 499998)** is **(list t 'q 'q f 'q 'q 'q t)**.

3.3 Pass 3

It is the job of pass 3 to eliminate the nondeterministic signals using the oracle. The function formalizing this pass is called **det** (for ''determine'').

Definition.

```
(det lst oracle)
   =
(if (nlistp lst)
    lst
    (if (equal (car lst) 'q)
        (cons (if (car oracle) t f)
              (det (cdr lst) (cdr oracle)))
        (cons (car lst)
              (det (cdr lst) oracle))))
```

The oracle parameter to our model is just an arbitrary list. The successive elements of the oracle are matched with the successive **'q**s in the list of signals to be processed, **lst**. Each oracle element specifies whether the corresponding **'q** should be replaced by **t** or by **f**.[6] **Det** merely copies the list of signals, replacing each **'q** as directed by the **oracle**.

3.4 Combining the Passes

Finally, to define **async** we compose the three passes.

Definition.

```
(async lst ts tr w r oracle)
   =
(det (warp (smooth t lst) ts tr w r)
     oracle)
```

Observe that we **smooth** the writer's view using **t** as the initial signal on the pin. This is an arbitrary choice.

4. The Biphase Mark Protocol

One use of a formal model of asynchrony is to investigate the circumstances under which communication protocols work properly. We illustrate such a use of our model by considering a biphase mark protocol. Recall Figure 3 where the protocol is informally described.

We will use an unbalanced configuration in which the mark subcell is just long enough to guarantee that it will be detected and the code subcell is just long enough to guarantee that the sweet spot is always sampled. See Figure 6.

---

[6]The axioms of the NQTHM logic define **car** and **cdr** to be non-**f** constants on non-**listp**s. The effect here is that if **oracle** is too short it is implicitly extended with as many **t**s as required.

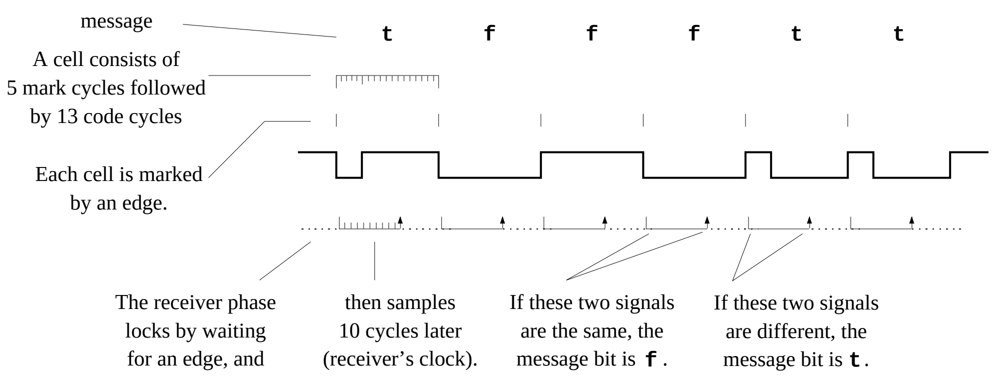


Figure 6: Our Modified Biphase Mark Protocol

In order to state a theorem about the protocol we must formalize it. In our formalization, the sizes of the two subcells and the sampling distance are parameters that are not fixed until we state the correctness theorem.

4.1 Sending

We will formalize the send side of the protocol by defining a function that maps from messages to signal streams, both of which are formally represented by bit vectors.

The fundamental notion in the protocol is that of the ''cell.'' Each cell is a list of **n**+**k** signals. Each cell encodes one bit, **b**, of the message, but the encoding depends upon the signal, **x**, output immediately before the cell. Let **x̄** be **(b-not x)**. Let **csig** be **(if b x x̄)**. Then a cell is defined as the concatenation of a ''mark'' subcell containing **n x̄**s followed by a ''code'' subcell containing **k csig**s.

Definition.
```
(cell x n k b)
   =
(app (listn n (b-not x))
     (listn k (if b x (b-not x)))).
```

Because **(if b x (b-not x))** reoccurs, it is convenient to define it as **(csig x b)**. Observe that the last signal in the cell is **(csig x b)**.

To encode a bit vector, **msg**, with cell size **n**+**k**, assuming that the previously output signal is **x** we merely concatenate successive cells, being careful to pass the correct value for the ''previous signal.''

Definition.
```
(cells x n k msg)
   =
(if (listp msg)
    (app (cell x n k (car msg))
         (cells (csig x (car msg))
                n k
                (cdr msg)))
    nil)
```

We adopt the convention that the sender holds the line high before and after the message is sent. Thus, on either side of the encoded cells we include ‘‘pads’’ of **t**, of arbitrary lengths **p1** and **p2**. The formal definition of **send** is

Definition.
```
(send msg p1 n k p2)
    =
(app (listn p1 t)
     (app (cells t n k msg) (listn p2 t))).
```

To send the message **(list t f t t)** with cells of size 1+2, padding the message at the front with three **t**s and at the back with five **t**s, we use **(send (list t f t t) 3 1 2 5)**. Its value is shown in Figure 7.

```
(send (list t f t t) 3 1 2 5)
   =
(list  t t t f t t f f f t f f t f f t t t t t )
```

Figure 7: Sending **(list t f t t)** with Cells of Size 1+2

4.2 Receiving

The receive side of the protocol will be formalized as a function from signal streams to messages. We need two auxiliary functions.

Scan takes a signal, **x**, and a list of signals, **lst**, and scans **lst** until it finds the first signal different from **x**. If **lst** happens to begin with a string of **x**s, **scan** finds the first edge.

Definition.
```
(scan x lst)
    =
(if (nlistp lst)
    nil
    (if (b-xor x (car lst))
        lst
        (scan x (cdr lst))))
```

For example, **(scan t (list t t t f f f t))** is **(list f f f t)**.

Recv-bit is the function that recovers the bit encoded in a cell. It takes two arguments. The first is the 0-based sampling distance, **j**, at which it is supposed to sample (e.g., if the cell length is 5+13, then **j** is 10). The second argument is the list of signals, starting with the first signal in the mark subcell of the cell.

Definition.
```
(recv-bit j lst)
    =
(if (b-xor (car lst) (nth j lst))
    t f)
```

The bit received is **t** if the first signal of the mark is different from the signal sampled in the code subcell;

otherwise, the bit received is **`f`**.

We can use **`scan`** and **`recv-bit`** to define the receive protocol. In our formalization, the receiver must know how many bits, **`i`**, to recover. In an actual application this might be a constant or it might have been transmitted earlier in a message of constant length. The list of signals on which **`recv`** operates should be thought of as starting with the signal, **`x`**, sampled in the code subcell of previous cell. If **`i`** is 0, the empty message is recovered. Otherwise, **`recv`** scans to the next edge (i.e., it scans past the initial **`x`**s to get past the code subcell of the previous cell and to the mark of the next cell). **`Recv`** then uses **`recv-bit`** to recover the bit in that cell and conses it to the result of recursively recovering **`i`**–1 more bits.

Definition.
```
(recv i x j lst)
   =
(if (zerop i)
    nil
    (cons (recv-bit j (scan x lst))
          (recv (sub1 i)
                (nth j (scan x lst))
                j
                (cdrn j (scan x lst)))))
```

Observe that in its recursive call, the new list of signals is the tail of **`lst`** that begins with the signal sampled by **`recv-bit`**. The new **`x`** is that signal.

To illustrate **`recv`**, let *lst* be the list produced by the **`send`** expression in Figure 7. Then **`(recv 4 t 2`** *lst***`)`** is the original message, **`(list t f t t)`**.

The phase locking is essentially implemented by **`scan`**. Observe that in all uses of **`lst`**, **`recv`** uses **`scan`** to find the first edge. Thus, no matter how many trailing signals there are in a cell (due to the different rates at which the two processors count), **`recv`** phase locks onto the beginning of the new cell. The clock rates are crucially important only from the time the cell is detected to the time the code subcell is sampled.

5. The Theorem

Do **`send`** and **`recv`** cope with the problems introduced by asynchrony? We can address this question formally now.

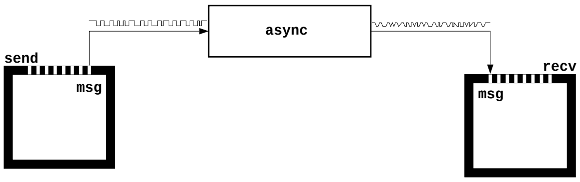


Figure 8: The Composition of **`send`**, **`async`** and **`recv`**

The diagram in Figure 8 suggests something we would like to prove about **`send`**, **`async`**, and **`recv`**: their composition is an identity. Of course, this is true only under certain assumptions, which we must make explicit. The composition we will study is

```
(recv (len msg)
      t 10
      (async (send msg p1 5 13 p2)
             ts tr w r oracle)).
```

We discuss this term from the inside out, making our assumptions clear.

(send msg p1 5 13 p2): We send some message **msg** in cells of size 5+13 with a leading pad of **p1 t**s and a trailing pad of **p2 t**s. We will require that **msg** be a bit vector but it can have arbitrary length. **P1** and **p2** are arbitrary (though, for technical reasons, we will require that the first one, at least, is a natural number).

(async (send ...) ts tr w r oracle): The signal stream generated by **send** is fed, in turn, to our model of asynchrony, which has the four clock parameters and the oracle as additional arguments. The model itself imposes certain constraints on the clock parameters: all are nonnegative integers and **ts** ≤ **tr** < **ts**+**w**. Those conditions put no limitation on the applicability of our result; it would still address arbitrarily clocked processors, arbitrary delay between them, and arbitrary phase displacement. However, some restrictions must be imposed to make the composition an identity. First, we must assume that the cycle times, **w** and **r**, are nonzero in order to avoid obvious pathological failures. Second, we must assume that the cycle times are ‘‘in close proximity,’’ which we will make precise by defining **(rate-proximity w r)**. The condition we wish to impose is $\frac{17}{18} \le \frac{\mathbf{w}}{\mathbf{r}} \le \frac{19}{18}$. But since we have limited ourselves to natural arithmetic, we define **rate-proximity** equivalently via

Definition.

```
(rate-proximity w r)
    =
(and
  (not (lessp (times 18 w) (times 17 r)))
  (not (lessp (times 19 r) (times 18 w)))).
```

We put no restrictions on **oracle**, thus addressing ourselves to all possible nondeterministic behaviors.

(recv (len msg) t 10 (async ...)): Finally, the output of our model is fed to the receiver. We impose no additional restrictions due to this term. But note that the first three arguments to **recv** limit the applicability of the theorem to cases in which we are trying to recover the correct number of bits of message, the line is initially high, and each cell is sampled 10 cycles after mark detection.

The theorem we will prove, named ‘‘BPM18’’ for ‘‘Biphase Mark, 18-cycles/bit,’’ is

Theorem. BPM18

```
(implies
 (and (bvp msg)
      (numberp ts)
      (numberp tr)
      (not (zerop w))
      (not (zerop r))
      (not (lessp tr ts))
      (lessp tr (plus ts w))
      (rate-proximity w r)
      (numberp p1))
 (equal (recv (len msg)
              t 10
              (async (send msg p1 5 13 p2)
                     ts tr w r oracle))
```

msg)).

The theorem would appear simpler had we built in the constants 10, 5 and 13 as well as the pad lengths, **`p1`** and **`p2`**, and the initial line value, **`t`**. We stated the theorem this way so it was convenient to experiment with different values.

The definition of **`rate-proximity`** forces $\frac{\mathtt{w}}{\mathtt{r}}$ to be within $\frac{1}{18}$ of unity. For what it is worth, $\frac{1}{18}$ is $0.0\overline{5}$, or somewhat more than 5%.

Formalization Revisited

Recall that the formalization problem can be cast as a programming problem:

Challenge: Write a Pure Lisp program (together with its subroutines) with the property that if the program returns **`t`** on all possible inputs then you will believe that the biphase mark protocol works.

BPM18 can be regarded as a Pure Lisp program that takes eight arguments: **`msg`**, **`ts`**, **`tr`**, **`w`**, **`r`**, **`p1`**, **`p2`**, and **`oracle`**. For specifically given values of those eight arguments it is straightforward to compute the value of the formula. The value will be **`t`** if the arguments satisfy the hypothesis and the conclusion is true or if the arguments fail to satisfy the hypothesis. The value will will be **`f`** otherwise.

If BPM18 is a theorem, then this program will return **`t`** on all inputs. Suppose it is a theorem. Do you believe that the biphase mark protocol always works under the hypothesis given? *That* is the formalization problem.

6. Formal Experiments

Before attempting to prove anything about **`send`** and **`recv`** we simply execute them to illustrate how they cope with **`async`**. Suppose we want to send the message **`(list t f t t)`**, using our 5+13 cycle protocol. To be concrete, we will precede the transmission with seven high cycles and follow it with eleven high cycles. The appropriate **`send`** expression is **`(send (list t f t t) 7 5 13 11)`**. A total of 90 write cycles are modeled in the output of this expression. The output is displayed graphically in Figure 9.

Now suppose the writer has a cycle time of 100, suppose the reader has a cycle time of 96, and suppose the first signal in the output arrives at the reader 30 time units before the reader's clock next ticks. Figure 9 shows (one of) the received waveforms. The **`oracle`** argument to **`async`** determines which of the waveforms is actually received. **`Recv`** must be able to cope with all of them. Observe that in this example, a total of 93 read cycles are modeled. The cells parsed by **`recv`** consume varying numbers of cycles. This variance is in part due to the slightly faster cycle time of the reader and in part to the nondeterministic choices on where the edges are located.

`Recv` correctly recovers the message **`(list t f t t)`** in this example.

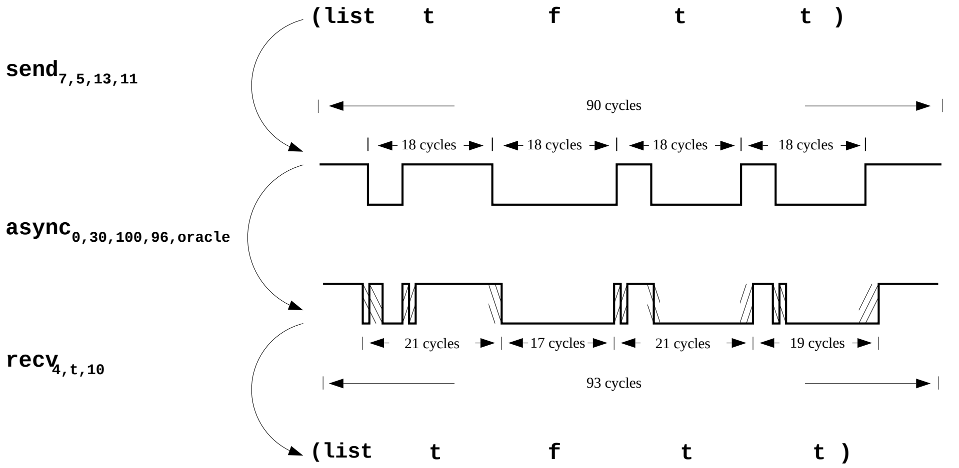


Figure 9: An Experiment with **send**, **async**, and **recv**

7. Proofs

BPM18 can be proved by transforming it into a slightly different form and then appealing to a more general theorem which we prove by induction. We give the proof later. We do not include in this paper the entire NQTHM transcript. Readers interested in the transcript should write the author. The transcript will reproduce the entire proof on the released version of the NQTHM theorem prover.

Our proof strategy is roughly as follows.

- We derive the shape of the **send** waveform after it has been processed by the first two passes of **async**, that is, we produce the ramped version of the received waveform. To do this we shall have to develop a body of lemmas about **async** and its subfunctions. We call this the ‘‘reusable theory’’ of **async** because it is independent of our particular application.
- We establish bounds on the lengths of each of the regions in the ramped waveform. This is basically a continuation of the reusable theory.
- We move into **recv** and show that scanning across a ramp nondeterministically defines a point in a region whose length is one larger than the ramp.
- Finally, this point is translated down the ramped waveform a fixed distance by **cdrn**, where it becomes the sampling point, and is shown to fall in the ‘‘sweet spot’’—that portion of the code subcell unaffected by ramps. This final step requires proving two key inequalities that establish that the sweet spot entirely contains the nondeterministically defined sampling point. These inequalities are proved by appealing to the bounds on the lengths of the various regions.

Because the message is of arbitrary length, all four of these steps are wrapped in an induction on the length of the message and are applied in turn to that portion of the wave generated in response to a single bit of the message.

7.1 The Reusable Theory of `Async`

7.1-A The Waveform Generators

While some steps in the proof are concerned with the peculiar properties of **`send`** and **`recv`**, most of the work is in establishing general properties of **`async`** and its interaction with the waveform primitives, **`app`** and **`listn`**.

In what sense are **`app`** and **`listn`** the ''waveform primitives?'' Informally, ideal signals are square waves; in our formalism, these square waves are generated by combinations of **`listn`** and **`app`** expressions—we use **`listn`**s to generate either ''high'' or ''low'' horizontal lines and then use **`app`**s to stick them together to form the vertical edges. As the signals get smoothed and warped in our model, the square corners become multivalued ramps; these ramps are formally generated by more **`listn`** expressions, only this time the signal repeated is **`'q`**. Thus, from the formal or algebraic point of view, the signal generators are **`app`** and **`listn`**. Because timing is crucial, we are also interested in the length, i.e., **`len`**, of such waveforms.

Given some input waveform, described formally, we would like to have enough symbolic machinery to allow us to derive the waveform produced by **`async`**. We would like both the input and the output waveforms to be described in terms of **`app`** and **`listn`**. Therefore, we seek a collection of theorems about **`app`**, **`listn`**, **`len`** and the three passes of **`async`**. Most of the theorems express distributivity laws, e.g., how to express the **`smooth`** of an **`app`** as the **`app`** of two **`smooth`**s. These theorems are independent of the particular signals generated by the biphase mark protocol. They are a first step toward what we call a ''reusable formal theory'' or ''rule book'' for **`async`**. They are only the first step because we stopped when we had enough rules to prove biphase mark correct.

7.1-B Elementary Rules

There are a variety of rules about **`app`** and **`listn`** that we here take for granted, though they were stated and proved in our mechanically checked work. We state a few as warm-up exercises.

Theorems.
`app` is associative:
```
(equal (app (app a b) c) (app a (app b c)))
```

`app` cancellation:
```
(equal (equal (app a b) (app a c))
       (equal b c))
```

`len` of **`app`**:
```
(equal (len (app a b)) (plus (len a) (len b)))
```

`len` of **`listn`**:
```
(implies (numberp n)
         (equal (len (listn n flg)) n))
```

fool's edge:
```
(equal (app (listn m flg) (listn n flg))
       (listn (plus m n) flg))
```

The last rule may bear explaining. Generally when we see the **`app`** of two **`listn`** expressions it describes an edge. But if the signal repeated by the first **`listn`** is the same as that repeated by the second, there is no edge and the **`app`** can be collapsed into a single **`listn`**. That is, if you draw a horizontal line at ‘‘high’’ followed by a horizontal line at ‘‘high’’ you get a (longer) horizontal line at ‘‘high.’’

We also assume all the usual theorems of integer arithmetic.

7.1-C Rules about Smooth

Suppose we are confronted by an application of **async** to the **app** of some **listn**s, i.e., we are trying to derive the shape of the waveform after **async** has mangled it. The definition of **async** can be expanded into a composition of **smooth**, **warp**, and **det**. If we can distribute these functions over **app** and **listn** we can derive the shape of the output. We treat **smooth** and **det** first and then turn to the much more complicated **warp**.

Recall that **smooth** takes as its first argument a Boolean flag, **flg1**, which is the ‘‘signal just previously passed’’ while smoothing a waveform supplied in the second argument. Some important theorems about **smooth** are shown below.

Theorems.

```
(equal (len (smooth flg lst)) (len lst)).

(implies (not (b-xor flg1 flg2))
         (equal (smooth flg1 (listn n flg2))
                (listn n flg2)))

(implies (and (b-xor flg1 flg2)
              (not (zerop n)))
         (equal (smooth flg1 (listn n flg2))
                (cons 'q (listn (sub1 n) flg2))))

(implies (not (b-xor flg1 flg2))
         (equal (smooth flg1
                        (app (listn n flg2) rest))
                (app (listn n flg2)
                     (smooth flg1 rest))))

(implies (and (b-xor flg1 flg2)
              (not (zerop n)))
         (equal (smooth flg1
                        (app (listn n flg2) rest))
                (app (smooth flg1 (listn n flg2))
                     (smooth flg2 rest))))
```

The first says that **smooth** does not change the **len** of the waveform. The second rule says that smoothing a list of **n** repetitions of **flg2** is a no-op if the signal just passed is Boolean-equivalent to **flg2**. I.e., no edge, no ramp. The third rule says that smoothing a list of **n** repetitions of **flg2** produces a ramp followed by **n**–1 repetitions of **flg2**, if the signal just passed is different from **flg2** and **n** is nonzero. The last two rules consider how to smooth a wave that starts with **n** repetions of **flg2** and then continues with some signals **rest**. If the signal just passed is equivalent to **flg2**, we can skip the smoothing of the initial segment and just smooth **rest**. If the signal just passed is different, we must smooth the initial segment (which, by the second rule, will produce a ramp) and then smooth **rest**, using **flg2** as signal just passed.

These theorems, and all other theorems displayed in this paper, have been proved mechanically with the NQTHM theorem prover (see box).

The NQTHM Theorem Prover

The NQTHM logic is supported by a mechanical theorem proving system [6]. The system enforces all the rules of the logic and also knows hundreds of heuristics for proving theorems in the logic. The user interacts with the system by submitting proposed definitions and theorems. The system checks each definition for admissibility and tries to prove each theorem. When it is successful, the theorem is processed into a ‘‘rule’’ and stored in a data base for future use.

The system’s proof attempts are driven by its heuristics and the rule base. When the system fails to find a proof, the user may guide it by submitting easier theorems that, when used as rules, lead the system to the proof it missed. To guide the theorem prover to the proof of a hard theorem the user must know a proof of the theorem and must understand how the system derives rules from theorems. In essence, the user programs the theorem prover in the art of proving particular kinds of theorems. Since the system must prove everything before using it, the user bears no responsibility for the correctness of proofs.

The system prints its proof as it goes. Users learn how to read these proofs so they know when the system is going down a blind alley. Here is the output produced for

```
Theorem. LEN-APP
(equal (len (app a b)) (plus (len a) (len b))).

Proof.
     Call the conjecture *1.

     Perhaps we can prove it by induction.  Three inductions are
suggested by terms in the conjecture.  They merge into two likely
candidate inductions.  However, only one is unflawed.  We will induct
according to the following scheme:
      (AND (IMPLIES (NLISTP A) (p A B))
           (IMPLIES (AND (NOT (NLISTP A)) (p (CDR A) B)) (p A B))).
Linear arithmetic, the lemmas CDR-LESSEQP and CDR-LESSP, and the
definition of NLISTP can be used to prove that the measure (COUNT A)
decreases according to the well-founded relation LESSP in each
induction step of the scheme.  The above induction scheme leads to
two new goals:

Case 2. (IMPLIES (NLISTP A)
                 (EQUAL (LEN (APP A B))
                        (PLUS (LEN A) (LEN B)))),
  which simplifies, opening up the definitions of NLISTP, APP, LEN,
  EQUAL, and PLUS, to:

        T.

Case 1. (IMPLIES (AND (NOT (NLISTP A))
                      (EQUAL (LEN (APP (CDR A) B))
                             (PLUS (LEN (CDR A)) (LEN B))))
                 (EQUAL (LEN (APP A B))
                        (PLUS (LEN A) (LEN B)))),
  which simplifies, applying PLUS-COMMUTES1, CDR-CONS, and PLUS-ADD1,
  and unfolding NLISTP, APP, and LEN, to:

        T.

     That finishes the proof of *1.  Q.E.D.

[0.0 0.6 0.3]
```

The system refers to rules by name, e.g., **`PLUS-COMMUTES1`**. **`LEN-APP`** is the name of the rule derived from this theorem. The time taken to do the proof is 0.6 seconds on a Sun Microsystems 3/60.

7.1-D Rules about Det

The crucial rules about **det** are shown below.

Theorems.
```
(equal (len (det lst oracle))
       (len lst))

(implies (boolp flg)
         (equal (det (listn n flg) oracle)
                (listn n flg)))

(equal (det (app lst1 lst2) oracle)
       (app (det lst1 oracle)
            (det lst2 (oracle* lst1 oracle))))
```

The first says that **det** does not change the length of the waveform. The second says that if **flg** is Boolean (in particular, if **flg** is not **'q**), then determining **(listn n flg)** with any oracle is a no-op, i.e., no ramps, no nondeterminacy. The third rule tells us we can distribute **det** over an **app**—but note that the theorem mentions a function we have not seen before, **oracle***. This function was defined precisely so that we could state the distributivity rule for **det** and **app**. Recall that **det cdr**s the oracle every time it sees a **'q** in its list of signals. Consider the oracle that **det** is using at the time it finishes processing **lst1** in **(app lst1 lst2)**: it is the original oracle **cdr**ed once for every **'q** in **lst1**. **Oracle*** is defined to be just that oracle.

Definition.
```
(oracle* lst oracle)
   =
(if (nlistp lst)
    oracle
    (if (equal (car lst) 'q)
        (oracle* (cdr lst) (cdr oracle))
        (oracle* (cdr lst) oracle)))
```

Observe that as we apply the distribution law to a right-associated nest of **app**s, the oracle argument becomes increasingly messy as calls of **oracle*** pile up. It turns out that we do not care. Since the oracle is arbitrary, the one returned by **oracle*** may as well be too. None of our theorems require us to investigate the structure of the oracle.

Before leaving this section, let us get a glimpse of where we are going. Suppose we have a term such as

(async (<u>send</u> msg p1 5 13 p2) ...).

Note that we have underlined **send** above. This is merely intended to draw the reader's eye to the term in question. By ''opening'' or ''expanding'' the definition of **send**—that is, replacing the call of **send** by its body and simplifying the result— we can expose the fact that it generates the leading pad with **app** and **listn**. Thus, **(async (<u>send</u> msg p1 5 13 p2) ...)** becomes

(<u>async</u> [app (listn p1 t) ...] ...).

Note that we have used square brackets to delimit the new material. These brackets should be read as parentheses. Note that we have also underlined a new focal point. By expanding the definition of **async** we see that it is a composition of **smooth**, **warp**, and **det**,

[det (warp (<u>smooth</u> t (app (listn p1 t) ...)) ...) ...].

We can distribute the **smooth** over the **app** and observe that there is no initial edge because the previous signal on the pin is assumed to be **t** and the waveform starts with a string of **t**s. Thus, our term becomes

(det (<u>warp</u> [app (listn p1 t) (smooth t ...)] ...) ...).

We have not yet shown how to distribute **warp** over **app** but we will. Unlike the other passes, **warp** may change the length of the waveform and we get

(<u>det</u> [app (listn *p1* **t) (warp (smooth t ...) ...)] ...),**

where *p1* is some expression involving **p1** and the clock parameters of **warp**. Finally, we can distribute the **det** over the **app** and then observe that since **t** is Boolean the **det** of **(listn** *p1* **t)** is **(listn** *p1* **t)**. The result is

(app (listn *p1* **t) (det (warp (smooth t ...) ...) ...)).**

We have succeeded in getting the initial pad of **t**s out of the sender, through the model (which may change its length), and into the jaws of the receiver!

7.1-E Rules about Warp

Recall that **warp** takes four clock parameters in addition to the list of signals to be processed. Those parameters are usually assumed to satisfy

Definition.
```
(clock-params ts tr w r)
    =
(and (numberp ts)
     (numberp tr)
     (not (lessp tr ts))
     (lessp tr (plus ts w))
     (not (zerop w))
     (not (zerop r))).
```

7.1-E(1) The Length of Warp. The number of signals coming out of **warp** is related to the number going in via

Theorem.
```
(equal (len (warp lst ts tr w r))
       (n* (len lst) ts tr w r)).
```

Note that we introduce an auxiliary function, **n***, to express the relationship. We could define **n*** algebraically. Under the assumption **(clock-params ts tr w r)**, we can show that **(n* n ts tr w r)** is $\left\lfloor \frac{\mathbf{n} \times \mathbf{w} - (\mathbf{tr} - \mathbf{ts})}{\mathbf{r}} \right\rfloor$. This fact will be useful when we need to bound the number of signals. But for our present purposes, it is easier to deal with a recursive definition of **n*** that mimics the way **warp** recurses.

Definition.
```
(n* n ts tr w r)
    =
(if (or (zerop r)
        (nendp n ts (plus tr r) w))
    0
    (add1 (n* (nlst+ n ts (plus tr r) w)
              (nts+ n ts (plus tr r) w)
              (plus tr r)
              w r)))
```

The number of signals in the output of **warp** depends on the number in the input, but not on the identities of the signals. Thus, we have chosen to make **n***'s first parameter be the number of input signals rather than the signals themselves. We therefore have to define auxiliary functions **`nendp`**, **`nlst+`**, and **`nts+`** which are analogous to **`endp`**, **`lst+`**, and **`ts+`** except that they take the length of the waveform (and, in the case of **`nlst+`**, return the length of the waveform that **`lst+`** returns). The definitions of these functions are exactly analogous to those of their counterparts and we omit them for brevity.

The proof of the theorem that **`n*`** is the length of **`warp`** is by an induction ‘‘unwinding’’ **`warp`** (see box). It requires the analogous lemmas connecting **`endp`** to **`nendp`**, **`lst+`** to **`nlst+`**, and **`ts+`** to **`nts+`**.

Theorems.

```
(equal (endp lst ts tr w)
       (nendp (len lst) ts tr w))

(equal (len (lst+ lst ts tr w))
       (nlst+ (len lst) ts tr w))

(equal (ts+ lst ts tr w)
       (nts+ (len lst) ts tr w))
```

Recursion and Induction

Recursion and induction are duals. The execution of a recursive definition proceeds by decomposing composite objects into simpler components until the answer is obvious. An inductive proof shows how the truth of a proposition is preserved as one uses simple objects to construct composite ones. This duality is often useful in discovering proofs of theorems about recursive functions. By choosing an induction that ‘‘unwinds’’ a recursive function, you can set up a base case in which the function computes the answer trivially and an induction case in which the induction hypothesis provides exactly the information you need to know.

For example, consider the following recursive prescription for deciding if i is an even number. If i is 0, the answer is ‘‘yes’’; if i is 1, the answer is ‘‘no’’; otherwise recursively ask whether $i-2$ is even.

Now consider the proposition: ‘‘if i is even and j is even, then $i+j$ is even.’’

Proof. Let us induct so as to unwind ‘‘i is even.’’

Base Case 0. Suppose i is 0. In this case, the proposition becomes ‘‘if 0 is even and j is even, then $0+j$ is even’’ which simplifies to the obvious truth ‘‘if j is even then j is even.’’

Base Case 1. Suppose i is 1. Here the proposition becomes ‘‘If 1 is even and ... then ...’’ but since 1 is not even (as the definition of even tells us) the proposition is true because its hypothesis is vacuous.

Inductive Case. Suppose i is not 0 and not 1. We may assume the proposition with $i-2$ replacing i. That is, our *Induction Hypothesis* is ‘‘if $i-2$ is even and j is even then $i-2+j$ is even.’’ We must prove

‘‘if i is even and j is even, then $i + j$ is even.’’

By the definition of even this is

‘‘if $i-2$ is even and j is even, then $(i + j)-2$ is even.’’

But by arithmetic this is

‘‘if $i-2$ is even and j is even, then $i-2 + j$ is even,’’

which is our induction hypothesis. **Q.E.D.**

7.1-E(2) Distributing Warp over Listn. If there are no ramps in the input to **warp** then the waveform passes through unchanged except for its length,

Theorem.
```
(implies (and (numberp n)
              (clock-params ts tr w r))
         (equal (warp (listn n flg) ts tr w r)
                (listn (n* n ts tr w r) flg))).
```

This theorem is proved by an induction that ''unwinds'' **(n* n ts tr w r)**.

7.1-E(3) Distributing Warp over App. **Warp** distributes over **app**,

Theorem.
```
(implies (clock-params ts tr w r)
         (equal (warp (app lst1 lst2) ts tr w r)
                (app (warp lst1 ts tr w r)
                     (warp (app (lst* lst1 ts tr w r) lst2)
                           (ts* lst1 ts tr w r)
                           (tr* lst1 ts tr w r)
                           w r))))).
```

Again we see that we have to define auxiliary concepts to express the theorem. As **warp** processes the list of signals, weakly decreasing the length of the list each step, **ts** and **tr** increase as cycles of lengths **w** and **r** are laid out against eachother. If the list of signals is **(app lst1 lst2)** then at some point **warp** will need to look at the first signal in **lst2**. At that point, **warp**'s **lst**, **ts** and **tr** parameters will have some values, *lst**, *ts** and *tr**. *Lst** will be some **cdr** of **(app lst1 lst2)**; in fact, it will be **(app** *lst1** **lst2)** where *lst1** is some **cdr** of **lst1**. *Lst1** will not necessarily be empty: it may contain many signals, just not enough to account for the entire read cycle from *tr** to *tr**+**r**. The functions **lst***, **ts*** and **tr*** compute *lst1**, *ts**, and *tr**.

Definition.
```
(lst* lst ts tr w r)
   =
(if (or (zerop r)
        (endp lst ts (plus tr r) w))
    lst
    (lst* (lst+ lst ts (plus tr r) w)
          (ts+ lst ts (plus tr r) w)
          (plus tr r)
          w r))
```

Observe that this definition is analogous to **warp**'s except that instead of building up the output waveform it just returns the value of the **lst** parameter when the recursion terminates. The definitions of **ts*** and **tr*** are analogous but return the final values of the **ts** and **tr** parameters. We also define **nlst***, **nts***, and **ntr***, the versions of these three functions that operate on the length of **lst** rather than on **lst** itself (and, in the case of **nlst***, return the length of the result returned by **lst***). For example,

Definition.
```
(nlst* n ts tr w r)
   =
(if (or (zerop r)
        (nendp n ts (plus tr r) w))
    n
    (nlst* (nlst+ n ts (plus tr r) w)
           (nts+ n ts (plus tr r) w)
           (plus tr r)
           w r))
```

We prove the obvious theorems about these functions and their counterparts, e.g., `(len (lst* lst ts tr w r))` is `(nlst* (len lst) ts tr w r)`.

The proof of the distribution law for `warp` over `app` is by an induction unwinding `(warp lst1 ts tr w r)`. The proof requires several analogous lemmas about how `endp`, `lst+`, `ts+` and `sig` handle `app`, e.g.,

Theorems.

```
(implies (and (not (endp lst1 ts tr+ w))
              (not (zerop w)))
         (not (endp (app lst1 lst2) ts tr+ w)))

(implies (and (not (endp lst1 ts tr+ w))
              (not (zerop w)))
         (equal (ts+ (app lst1 lst2) ts tr+ w)
                (ts+ lst1 ts tr+ w)))
```

7.1-E(4) Warping in the Vicinity of a Ramp. When we distribute `warp` over `(app lst1 lst2)` we get two `warp` expressions. The first one is simply the `warp` of `lst1`. But the second one, which intuitively is the `warp` of `lst2`, actually depends on what happens as `warp` crosses the ‘‘gap’’ between `lst1` and `lst2`. An example makes this clear.

Suppose that the input waveform is `(app (listn n t)` *rest*`)`. The distribution law tells us that the `warp` is

```
(app (warp (listn n t) ts tr w r)
     (warp (app (lst* (listn n t) ts tr w r) rest)
           (ts* (listn n t) ts tr w r)
           (tr* (listn n t) ts tr w r)
           w r)).
```

Of course, we know that `(warp (listn n t) ts tr w r)` is `(listn (n* n ts tr w r) t)` and so the initial part of the emerging waveform is known. But we cannot yet see how *rest* emerges because we have not driven `warp` across the gap. The last few remaining signals in the first part must be processed in conjunction with the first few signals of *rest*. One read cycle spans this gap, the one that starts at the time computed by `(tr* (listn n t) ts tr w r)`.

Because `warp` is used exclusively after `smooth` (and the use of the fool’s edge rule), the first signal after a string of `t`s (or `f`s) will be a ramp. That ramp will necessarily participate in the reconciliation of the signals arriving during the read cycle identified by `tr*`. It may influence several read cycles, causing `warp` to produce a string of ramps. If `w` and `r` are within a factor of 2 of eachother, i.e., neither cycle time is long enough to completely contain two cycles of the other processor, then a single ramp coming into `warp` can produce 1, 2, or 3 ramps coming out. We illustrate the three possibilities in Figure 10.

In Figure 10 we show how `warp` passes through the ramp in `(app (listn n t) (cons 'q rest))` in three different contexts involving how the two series of cycles overlap. Recall that the read cycle that starts at `tr*` is, by definition, the first read cycle that is influenced by the ramp. That cycle may also be influenced by the last few signals in the first part of the incoming waveform, in this case, the last few `t`s of `(listn n t)`. `Lst*` is, by definition, those last few signals— the signals preceding the ramp that must be reconciled with the ramp. `Ts*` is the arrival time of the first signal in `lst*`.

In Configuration A, the read cycle that starts at `tr*` entirely consumes the ramp. In the picture, we show `lst*` as containing one signal, the last one preceding the ramp. This is just one of two possibilities. It is

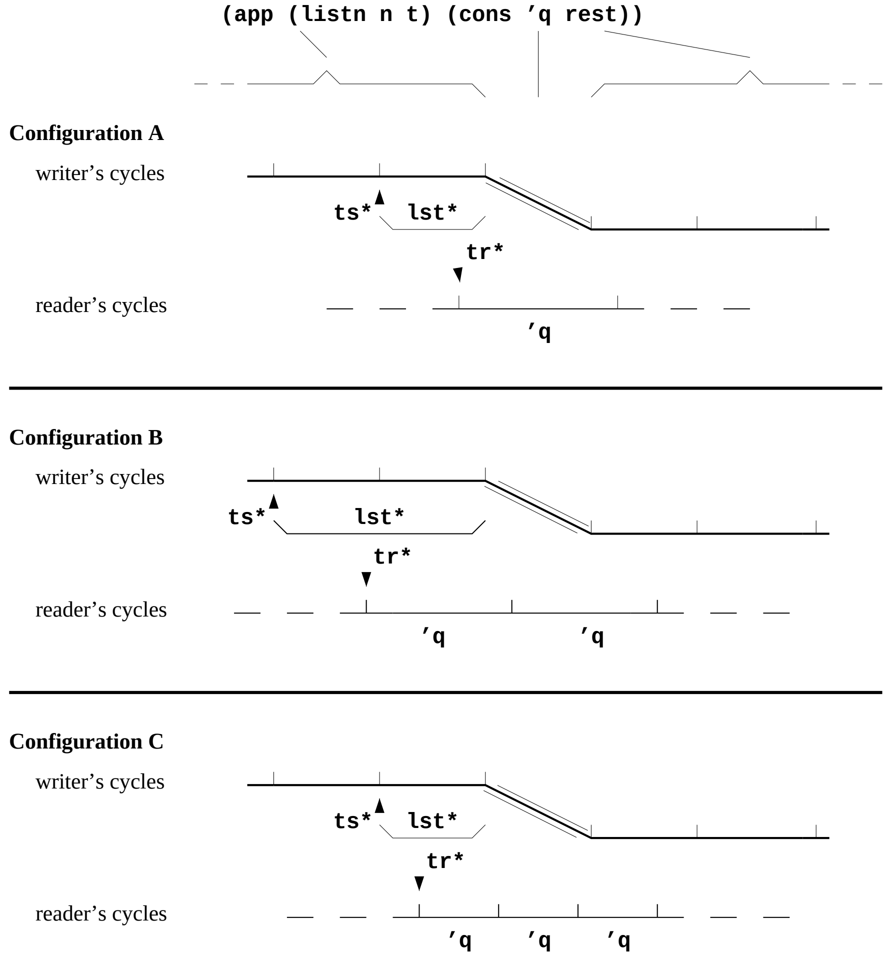


Figure 10: Warping Across a Ramp

possible for **`lst*`** to be empty (i.e., for the signals preceding the ramp to determine a whole number of read cycles). In Configuration B, the read cycle at **`tr*`** splits the ramp so that it falls into two read cycles. Again, our picture shows one of several possibilities regarding **`lst*`**: it contains the last two signals preceding the ramp here, but if the reader's cycle time were shorter it could contain only the last signal or no signals. Finally, in Configuration C, the read cycles fall so that the one after **`tr*`** is entirely consumed by the ramp. If the read cycle can be arbitrarily shorter than the write cycle, an arbitrary number of **`'q`**s might emerge from **`warp`** due to a single ramp. But in our reusable theory about **`warp`** we chose to limit our attention to processors whose cycle times are within a factor of 2 of eachother.

Once **`warp`** has gotten past the ramp, how many signals remain in **`rest`** to process? In the three configurations illustrated, the first signal in **`rest`** is always involved in the determination of the first read cycle after passing the ramp. But this need not be the case. To see why, consider Configuration B and slide the reader's cycles down about half a write cycle, so that **`lst*`** now contains only one element and the first signal of **`rest`** is consumed in emitting the second **`'q`**. Under our ''factor of 2'' assumption, there are only two cases: no signals from **`rest`** are consumed in passing the ramp or one signal is consumed.

The observations just made can be combined with our previously proved theorems about **warp**, **app**, and **listn** to derive the following extremely useful rule.

Theorem.
```
(implies (and (clock-params ts tr w r)
              (lessp w (times 2 r))
              (lessp r (times 2 w))
              (numberp n)
              (lessp 2 (len rest)))
         (equal (warp (app (listn n flg) (cons 'q rest))
                      ts tr w r)
                (app (listn (n* n ts tr w r) flg)
                     (app (listn (nq n ts tr w r) 'q)
                          (warp (cdrn (dw n ts tr w r) rest)
                                (ts n ts tr w r)
                                (tr n ts tr w r)
                                w r))))))
```

The functions **nq**, **dw**, **ts**, and **tr** will be discussed below, but first let us consider this theorem. It addresses itself to warping across a ramp in the general case where the ramp is preceded by an arbitrarily long stable signal. The clock rates must be within a factor of 2 and there must be at least one signal after the ramp. The theorem tells us that the emerging signal is composed of three parts. First, we get the warped image of the long stable signal (possibly stretched or shrunk), i.e., **(listn (n* n ts tr w r) flg)**. Then we get a certain number of ramps, **(listn (nq n ts tr w r) 'q)**, where the function **nq** tells us how many. **Nq**, defined below, is either 1, 2, or 3. Finally we get the warped image of some **cdr** of **rest**. The function **dw** determines how many signals are chopped off of **rest** and its value is either 0 or 1. The values of **ts** and **tr** used while warping the rest are computed by the functions **ts** and **tr**. This is a beautiful result because it formalizes and makes precise the claim that warp stretches or shrinks the waveform without altering its basic shape.

The proof of the above theorem is tedious but straightforward, given our prior results. The general theorem for warping an **app** applies and, together with the theorem for warping a **listn**, explains the **(listn (n* n ts tr w r) flg)** of the result. We then are left with warping across the ramp and into **rest**. We do that entirely with case analysis as suggested by Figure 10.

Here are the definitions of the new functions used in the theorem. We only explain the first, **nq** and its subroutine **nqg**. The others are similar. The task of **nq** is to compute the number of **'q**s emitted by **warp** in response to a single ramp preceded by **n** identical signals. Warping across those **n** signals, starting at **ts** and **tr**, will bring us to some tail, which we have called *lst1**, whose signals must be reconciled with the ramp. The **tr** and **ts** parameters of **warp** at that point are **(nts* n ts tr w r)**, abbreviated as *ts* below, and **(ntr* n ts tr w r)**, abbreviated as *tr* below. *Lst1** is of length **(nlst* n ts tr w r)**, which we henceforth abbreviate as *k*. The ramp is processed with those *k* signals, starting at *ts* and *tr*. It influences every read cycle that intersects with it, causing each to be nondeterminate. How many are there? That depends on how successive cycles fall. For example, if *tr*+**r** (the start of the next read cycle) is greater than or equal to *ts*+*k***w**+**w** (the arrival time of the signal after the ramp), then only one read cycle is influenced. Continued case analysis leads to the following definition of **nq**.

Definition.
```
(nq n ts tr w r)
   =
(nqg (nlst* n ts tr w r)
     (nts* n ts tr w r)
     (ntr* n ts tr w r)
     w r)
```

Definition.
```
(nqg k ts tr w r)
   =
(if (lessp (plus r tr)
           (plus w ts (times w k)))
    (if (lessp (plus r r tr)
               (plus w ts (times w k)))
        3 2)
```

```
1)
```

Summarizing, the **nlst***, **nts***, and **ntr*** expressions in **nq** determine the parameters **warp** has when it first has to process the ramp, and **nqg** takes those parameters and does a case analysis to determine how many **'q**s are emitted before getting past the ramp.

The definitional styles of **dw**, **ts**, and **tr** are identical and we display them without comment. Of course, the case analysis in each is unique.

Definition.
```
(dw n ts tr w r)
   =
(dwg (nlst* n ts tr w r)
     (nts* n ts tr w r)
     (ntr* n ts tr w r)
     w r)
```

Definition.
```
(dwg k ts tr w r)
   =
(if (lessp (plus r tr)
           (plus ts w (times k w)))
    (if (lessp (plus r r tr)
               (plus ts w w
                     (times k w)))
        0 1)
(if (equal (plus r tr)
           (plus ts w (times k w)))
    0
(if (and (lessp (plus ts w (times k w))
                (plus r tr))
         (lessp (plus r tr)
                (plus ts w w
                      (times k w))))
        0
(if (lessp (plus r tr)
           (plus ts w w (times k w)))
    0 1))))
```

Definition.
```
(ts n ts tr w r)
   =
(tsg (nlst* n ts tr w r)
     (nts* n ts tr w r)
     (ntr* n ts tr w r)
     w r)
```

Definition.
```
(tsg k ts tr w r)
   =
(plus ts
      (times w k)
      w
      (times w (dwg k ts tr w r)))
```

Definition.
```
(tr n ts tr w r)
   =
(trg (nlst* n ts tr w r)
     (nts* n ts tr w r)
     (ntr* n ts tr w r)
     w r)
```

Definition.
```
(trg k ts tr w r)
   =
(plus tr (times r (nqg k ts tr w r)))
```

Of course, the accuracy of this case analysis is questionable until the theorem showing how **warp** processes a ramp is proved.[7]

It should be observed that **(clock-params ts tr w r)** implies **(clock-params (ts n ts tr w r) (tr n ts tr w r) w r)**, provided **n** is numeric and **w** and **r** are within a factor of 2 of eachother.

This completes our development of the reusable theory of **async**.

[7]Indeed, we did the analysis incorrectly many times before finally producing the correct one.

7.2 Bounding Certain Functions

The reusable theory introduces the functions **nq**, **dw**, and **n*** which are used in the determination of the lengths of various parts of the received waveform. It is useful in our coming proof to establish bounds for these functions.

7.2-A Bounding nq

Nq is the width of the nondetermistic region caused by warping a single ramp. From the definition of **nq** and its subfunction **nqg** it is obvious that 1 ≤ **(nq n ts tr w r)** ≤ 3.

7.2-B Bounding dw

Dw is the number of signals consumed by **warp** immediately after a single ramp. From the definition of **dw** and its subroutine **dwg**, it is obvious that 0 ≤ **(dw n ts tr w r)** ≤ 1.

7.2-C Bounding n*

N* is the length of the result of warping a horizontal region of the waveform. That is, **(warp (listn n flg) ts tr w r)** is **(listn (n* n ts tr w r) flg)**. Under the assumption **(clock-params ts tr w r)**, we can show that **(n* n ts tr w r)** is $\left\lfloor \frac{n \times w - (tr - ts)}{r} \right\rfloor$. This algebraic expression of **n*** can be proved by an induction unwinding **(n* n ts tr w r)** and using properties of **nendp** and **nts+** and natural number arithmetic.

We are interested in bounds on **n***. However, to derive interesting bounds we must impose some constraints on the cycle times **w** and **r** since otherwise **(n* n ts tr w r)** can be arbitrarily larger or smaller than **n**. Because we are headed toward the BPM18 theorem, where we assume **(rate-proximity w r)**, i.e., that 18 ticks of length **w** is between 17 and 19 ticks of length **r**, we investigate the bounds on **n*** in that context.

The following two theorems are fairly straightforward applications of the algebraic identity above and the usual properties of natural number arithmetic.

Theorem. **N***-lower-bound
```
(implies (and (clock-params ts tr w r)
              (rate-proximity w r)
              (numberp n)
              (lessp n 18))
         (not (lessp (n* n ts tr w r)
                     (sub1 (sub1 n)))))
```

Theorem. **N***-upper-bound
```
(implies (and (clock-params ts tr w r)
              (rate-proximity w r)
              (numberp n)
              (lessp n 18))
         (not (lessp n (n* n ts tr w r))))
```

Roughly speaking, if **w** and **r** are in proximity and **n**<18, then

n–2 ≤ **(n* n ts tr w r)** ≤ **n.**

In a truly general reusable theory of **async** we would derive bounds theorems that did not refer to the particular notion of proximity used in the intended application.

7.3 Scanning across a Ramp

Using the general theory of **async** we will be able to derive the form of the wave reaching the receiver. We now consider how the receiver reacts.

Because of

Theorem.
```
(equal (scan flg (app (listn n flg) rest))
       (scan flg rest))
```

it is easy to prove by induction that

Theorem.
```
(equal (recv n flg k (app (listn m flg) rest))
       (recv n flg k rest)).
```

That is, the receiver simply ignores the leading signals it is scanning past.

Eventually then the receiver will be confronted with a ramp of some arbitrary length followed by a region of parity opposite that being scanned. That is, the receiver will be confronted with a nondeterministically defined edge. We need to say how **scan** deals with such an edge. To state the desired lemma we have to define two new auxiliary functions, **no** and **scan-oracle**. Their definitions are obvious from the lemma we prove,

Theorem.
```
(implies (and (lessp 0 n)
              (b-xor flg1 flg2))
         (equal (scan flg1 (app (det (listn nq ’q) oracle)
                                (app (listn n flg2) rest)))
                (app (det (listn (no flg1 nq oracle)
                                 ’q)
                          (scan-oracle flg1 nq oracle))
                     (app (listn n flg2) rest)))).
```

Observe that on the left-hand side of the conclusion, **scan** is scanning past **flg1** and has encountered a ramp of length **nq** followed by a nonempty region of parity opposite that of **flg1**. Where does **scan** stop? The right-hand side tells us: it stops and returns a ramp of length **(no flg1 nq oracle)** followed by everything past the ramp. The term **(no flg1 nq oracle)** is defined to be the number of cycles in a ramp of length **nq** after the first one at which **oracle** differs from **flg1**. It is an easy consequence of the definition of **no** that **0** ≤ **(no flg1 nq oracle)** ≤ **nq**. **(Scan-oracle flg1 nq oracle)** is the remaining tail of **oracle** as of the cycle in question.

Of particular interest is the length of the waveform **scan** returns: it is just **(no flg1 nq oracle)** + **n** + **(len rest)**.

7.4 Finding the Sampling Point

Once **recv** has used **scan** to find the next edge (which is always nondeterministically defined), it uses **cdrn** to wait the specified number of cycles. This nondeterministically defines the sampling point. Formally, we need a lemma which distributes **cdrn** down a waveform.

A particularly nice rule is

Theorem.
```
(equal (cdrn dw (listn n flg))
```

```
        (listn (difference n dw) flg)),
```

which says that waiting **dw** cycles on a flat wave of length **n** produces a flat wave of length **n**–**dw**. Here **difference** is the ‘‘nonnegative difference’’ function that returns 0 if the subtrahend is larger than the minuend.

To distribute **cdrn** over an **app** we use the equally elegant

Theorem.

```
(equal (cdrn n (app a b))
       (if (lessp n (len a))
           (app (cdrn n a) b)
           (cdrn (difference n (len a)) b))).
```

Thus, waiting **n** cycles on a wave composed of two parts, **a** and **b** is the same thing as waiting on **a** and then concatenating **b** or waiting for fewer cycles on **b** alone, depending on whether **a** is sufficiently long to survive the waiting.

This completes our strategic development.

7.5 The Proof of BPM18

Theorem. BPM18

```
(implies
 (and (bvp msg)
      (numberp ts)
      (numberp tr)
      (not (zerop w))
      (not (zerop r))
      (not (lessp tr ts))
      (lessp tr (plus ts w))
      (rate-proximity w r)
      (numberp p1))
 (equal (recv (len msg) t 10
              (async (send msg p1 5 13 p2)
                     ts tr w r oracle))
        msg))
```

Proof. We transform the left-hand side of the conclusion into a slightly different form and then appeal to a lemma (proved afterwards by induction). First, observe that the theorem is trivial if **msg** is empty: **(len msg)** is 0 and hence **recv** returns **nil**, which is equal to **msg**. Thus, we may assume **msg** is a **listp**. Now consider the left-hand side above

```
(recv (len msg) t 10
      (async (send msg p1 5 13 p2)
             ts tr w r oracle)).
```

By expanding the definitions of **send** and **async** we get

```
(recv (len msg) t 10
      [det
       (warp
        (smooth t
                (app (listn p1 t)
                     (app (cells t 5 13 msg)
                          (listn p2 t))))
        ts tr w r)
```

```
       oracle]).
```

We now focus on the **app** underlined above. We know its argument is a **listp** whose **car** is **f** (because **cells** is passed the flag **t** and a non-**nil** **msg**). So we can expand **app** to get

```
(recv (len msg) t 10
      (det
       (warp
        (smooth t
                (app (listn p1 t)
                     [cons f
                           (app (cdr (cells t 5 13 msg))
                                (listn p2 t))]))
        ts tr w r)
       oracle)).
```

Now we distribute the **smooth**,

```
(recv (len msg) t 10
      (det
       (warp
        [app (listn p1 t)
             (cons 'q
                   (smooth f
                           (app (cdr (cells t 5 13 msg))
                                (listn p2 t))))]
        ts tr w r)
       oracle)).
```

Observe that the initial string of **t**s come through unscathed but the **f** turns into a ramp and the **smooth**, with its flag negated, finally nestles against the final **app**.

Focusing now on the **warp** term above, we drive it through the **app** and past the ramp,

```
(recv (len msg) t 10
      (det
       [app (listn (n* p1 ts tr w r) t)
            (app (listn (nq p1 ts tr w r) 'q)
                 (warp (cdrn (dw p1 ts tr w r)
                             (smooth f
                                     (app (cdr (cells t 5 13 msg))
                                          (listn p2 t))))
                       (ts p1 ts tr w r)
                       (tr p1 ts tr w r)
                       w r))]
       oracle)),
```

and then distribute **det** down the waveform

```
(recv (len msg) t 10
      [app (listn (n* p1 ts tr w r) t)
           (app (det (listn (nq p1 ts tr w r) 'q)
                     oracle)
                (det (warp (cdrn (dw p1 ts tr w r)
                                 (smooth f
                                         (app (cdr (cells t 5 13 msg))
                                              (listn p2 t))))
                           (ts p1 ts tr w r)
                           (tr p1 ts tr w r)
                           w r)
```

```
                    (oracle* (listn (nq p1 ts tr w r) 'q)
                             oracle)))]).
```

Note that the first **listn** term above survives the **det** unscathed: no ramp, no nondeterminacy. But that **listn** term is just a string of **t**s in the maw of a receiver scanning past **t**. So the above is equal to the result of removing that **listn**,

```
(recv (len msg) t 10
      (app (det (listn (nq p1 ts tr w r) 'q) oracle)
           (det (warp (cdrn (dw p1 ts tr w r)
                            (smooth f
                                   (app (cdr (cells t 5 13 msg))
                                        (listn p2 t))))
                      (ts p1 ts tr w r)
                      (tr p1 ts tr w r)
                      w r)
                (oracle* (listn (nq p1 ts tr w r) 'q)
                         oracle)))).
```

Inspection will show that the **recv** expression above is an instance of the more general one in our key BPM18-Lemma below. That lemma establishes that the **recv** returns **msg**. **Q.E.D.**

The form of our general lemma may be obtained by replacing certain terms above by variables. The **t** in the second argument of **recv** and the **f** in the first argument of **smooth** are replaced by arbitrary Boolean flags of opposite parity. The **nq**, **dw**, **ts** and **tr** terms are replaced by variables, constraining the **nq** replacement to be between 1 and 3, the **dw** replacement to be 0 or 1, and the **ts** and **tr** replacements to be **clock-params**. Finally, the **oracle*** expression is replaced by an arbitrary second oracle. The general lemma is shown below.

Theorem. BPM18-Lemma

```
(implies (and (bvp msg)
              (clock-params ts tr w r)
              (rate-proximity w r)
              (numberp nq)
              (not (lessp 3 nq))
              (numberp dw)
              (not (lessp 1 dw))
              (boolp flg1)
              (boolp flg2)
              (b-xor flg1 flg2))
         (equal (recv (len msg)
                      flg1
                      10
                      (app (det (listn nq 'q) oracle1)
                           (det (warp
                                  (cdrn dw
                                   (smooth flg2
                                    (app (cdr (cells flg1 5 13 msg))
                                         (listn p2 t))))
                                  ts tr w r)
                                 oracle2)))
                msg))
```

BPM18-Lemma describes the receiver in its general configuration rather than in its initial configuration. Two points bear noting. First, the unusual initial pad is gone: the receiver is processing a warped sequence of cells and is standing immediately in front of a blurred edge spread over **nq** cycles. Second, the receiver is scanning for an arbitrary edge as specified by **`flg1`** rather than just a falling one as required in the top-level application. This is particularly important because we will need to use our inductive hypothesis to process cells of parity opposite that of the first cell.[8]

The proof of BPM18-Lemma is by induction on the length of **`msg`**. We give the proof below but make some strategic remarks first. We separate two base cases, one for the empty **`msg`** and one for **`msg`**s of length 1. Thus, in the induction case the message is of length 2 or more and we therefore know the first cell is followed by another cell and, hence, by an edge. (The last cell is not necessarily followed by an edge.) That trailing edge in the induction conclusion will become the leading edge in the induction hypothesis.

There are two crucial points in the proof. (1) Does the receiver recover the bit in the first cell correctly? And (2), when it scans past the remains of that first cell, is the receiver back in the general situation described by our lemma, i.e., can we use our induction hypothesis? The answers to both questions hinge on certain arithmetic inequalities that tell us that the receiver is in the sweet spot of the waveform 10 cycles after detecting the leading edge. When we get around to answering these questions, we will consider the two subcases: was the first bit of **`msg`** a **`t`** or an **`f`**? Because the sweet spot for a cell encoding **`t`** is narrower than that for a cell encoding **`f`**, our discussion of the proof will be detailed only for the case of **`t`**. See Figure 11.

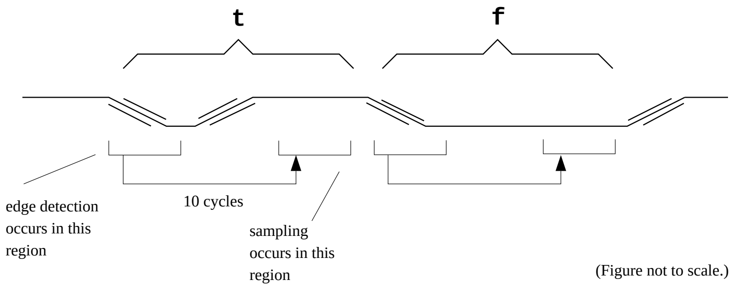


Figure 11: Is the Correct Bit Recovered?

Proof of BPM18-Lemma. The proof is by induction on the length of **`msg`**.

Base Case 0: If **`msg`** is not a **`listp`**, the proof is trivial because **`recv`** returns **`nil`**.

Base Case 1: Suppose **`(listp msg)`** and **`(nlistp (cdr msg))`**. Then **`msg`** = **`(list b)`** for some Boolean **`b`**. The proof of this case requires the full analysis of how **`b`** is encoded, smoothed, warped, determined, and recovered. This analysis will establish that we recover the correct bit and that the receiver is properly positioned to scan past the remaining signals in the sweet spot. The same analysis is used (but not described) in the induction step—our description of the induction case focus entirely on using the induction hypothesis.

[8] We do not have time to expound upon the subtleties of the BPM18-Lemma formula, but finding a statement of this lemma suitable for induction was the most creative part of the exercise.

We divide base case 1 into two subcases, according to whether **b** is **t** or **f**. We will describe only the case where **b** is **t**.

Consider the left-hand side of the conclusion of BPM18-Lemma, with **msg** replaced by **(list t)**,

lhs:

```
(recv (len (list t)) flg1 10
      (app (det (listn nq ’q) oracle1)
           (det
            (warp
             (cdrn dw
                   (smooth flg2
                           (app (cdr (cells flg1 5 13 (list t)))
                                (listn p2 t))))
             ts tr w r)
            oracle2))).
```

We wish to show that *lhs* is equal to **(list t)**. By expanding the definitions of **len**, **cells**, **cell** and **csig**, and using properties of **app** and **listn**, *lhs* is equal to

```
(recv 1 flg1 10
      (app (det (listn nq ’q) oracle1)
           (det
            (warp
             (cdrn dw
                   (smooth flg2
                           [app (listn 4 (b-not flg1))
                                (app (listn 13 flg1)
                                     (listn p2 t))]))
             ts tr w r)
            oracle2))).
```

Distributing **smooth** produces

```
(recv 1 flg1 10
      (app (det (listn nq ’q) oracle1)
           (det
            (warp
             (cdrn dw
                   [app (listn 4 (b-not flg1))
                        (cons ’q
                              (app (listn 12 flg1)
                                   (smooth flg1 (listn p2 t))))])
             ts tr w r)
            oracle2))).
```

Driving the **cdrn** into the **app** and then into its first argument produces

```
(recv 1 flg1 10
      (app (det (listn nq ’q) oracle1)
           (det
            (warp
             [app (listn (difference 4 dw) (b-not flg1))
                  (cons ’q
                        (app (listn 12 flg1)
                             (smooth flg1 (listn p2 t))))]
             ts tr w r)
            oracle2))).
```

This allows us to distribute **warp** over the **app** and the ramp. The result is

```
(recv 1 flg1 10
      (app (det (listn nq ’q) oracle1)
           (det
            [app (listn (n* (difference 4 dw) ts tr w r)
                        (b-not flg1))
                 (app (listn (nq (difference 4 dw) ts tr w r)
                             ’q)
                      (warp
                       (cdrn (dw (difference 4 dw) ts tr w r)
                             (app (listn 12 flg1)
                                  (smooth flg1 (listn p2 t))))
                       (ts (difference 4 dw) ts tr w r)
                       (tr (difference 4 dw) ts tr w r)
                       w r))]
            oracle2))).
```

Once again we drive the **cdrn** into the **app** and **listn**, producing

```
(recv 1 flg1 10
      (app (det (listn nq ’q) oracle1)
           (det (app (listn (n* (difference 4 dw) ts tr w r)
                            (b-not flg1))
                     (app (listn (nq (difference 4 dw) ts tr w r)
                                 ’q)
                          (warp
                           [app (listn (difference 12
                                                   (dw (difference 4 dw)
                                                       ts tr w r))
                                       flg1)
                                (smooth flg1 (listn p2 t))]
                           (ts (difference 4 dw) ts tr w r)
                           (tr (difference 4 dw) ts tr w r)
                           w r)))
                oracle2))),
```

to which we can apply **warp** distributivity again to produce

```
(recv 1 flg1 10
      (app (det (listn nq ’q) oracle1)
           (det (app (listn (n* (difference 4 dw) ts tr w r)
                            (b-not flg1))
                     (app (listn (nq (difference 4 dw) ts tr w r)
                                 ’q)
                          [app
                           (listn (n* (difference 12
                                                  (dw (difference 4 dw)
                                                      ts tr w r))
                                          (ts (difference 4 dw) ts tr w r)
                                          (tr (difference 4 dw) ts tr w r)
                                          w r)
                                      flg1)
                           warp]))
                oracle2))),
```

where *warp* is a complicated **warp** expression in which we are uninterested because it concerns the warping of the waveform after the code subcell in question.

Finally, we distribute **det** over the **app**s and **listn**s to get

```
(recv 1 flg1 10
      (app (det (listn nq 'q) oracle1)
           [app (listn (n* (difference 4 dw) ts tr w r)
                       (b-not flg1))
                (app (det (listn (nq (difference 4 dw) ts tr w r)
                                 'q)
                          oracle3)
                     (app (listn (n* (difference 12
                                                 (dw (difference 4 dw)
                                                     ts tr w r))
                                     (ts (difference 4 dw) ts tr w r)
                                     (tr (difference 4 dw) ts tr w r)
                                     w r)
                                 flg1)
                          (det warp oracle4)))])),
```

where *oracle3* and *oracle4* are **`oracle*`** expressions in which we are uninterested.

We find the printed form of this derivation extremely tedious. We remind the reader that during the original construction of this proof, all of these manipulations were done by NQTHM. The user's job was to state the lemmas.

Let us adopt the following abbreviations:

nq1
```
nq
```

s1
```
(n* (difference 4 dw) ts tr w r)
```

nq2
```
(nq (difference 4 dw) ts tr w r)
```

s2
```
(n* (difference 12
                (dw (difference 4 dw) ts tr w r))
    (ts (difference 4 dw) ts tr w r)
    (tr (difference 4 dw) ts tr w r)
    w r)
```

rest
```
(det warp oracle4)
```

waveform
```
(app (det (listn nq1 'q) oracle1)
     (app (listn s1 (b-not flg1))
          (app (det (listn nq2 'q) oracle3)
               (app (listn s2 flg1)
                    rest))))
```

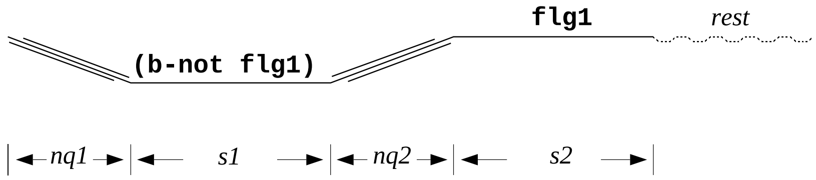


Figure 12: *Waveform* if **`flg1`** is ''High''

If we depict **flg1** as ‘‘high’’ and **(b-not flg1)** as ‘‘low’’, then *waveform* is as shown in Figure 12.

From our bounds discussion we can obtain upper and lower bounds on the lengths of each region of *waveform*. The bounds are shown in Figure 13. It should be noted that the bounds are all independently derived, i.e., it is not the case that all four quantities can simultaneously attain their extreme values, though we do not use this unproved observation. We return to this point later.

$$1 \le nq1 \le 3$$

$$1 \le s1 \le 4$$

$$1 \le nq2 \le 3$$

$$9 \le s2 \le 12$$

Figure 13: Bounds on the Lengths of Waveform Regions

We now resume our simplification of *lhs*, which, using our abbreviations, is just **(recv 1 flg1 10** *waveform***)**. By expanding the definition of **recv**, **recv-bit**, and **nth**, we see that *lhs* is equal to

```
(list (b-xor (b-not flg1)
             (car (cdrn 10 (scan flg1 waveform)))))
```

The important subterm is **(cdrn 10 (scan flg1** *waveform***))**. We claim that this term is equal to **(app (listn** *s2’* **flg1)** *rest***)**, where *s2’* is nonzero. Suppose this claim is true. Then the **car** of the **cdrn** is **flg1** and hence *lhs* is **(list (b-xor (b-not flg1) flg1))** which is **(list t)** and we are done.

Furthermore, looking ahead to the inductive case, this same **cdrn**/**scan** term is involved twice in the recursive call of **recv**. It is the waveform from which all other bits of the message will be recovered and its **car** is the signal **recv** will scan past while looking for the next edge. Proving that it is equal to **(app (listn** *s2’* **flg1)** *rest***)** will not only complete this base case but go a long way toward simplifying the inductive case.

Therefore, consider **(cdrn 10 (scan flg1** *waveform***))** where *waveform* is

```
(app (det (listn nq1 'q) oracle1)
     (app (listn s1 (b-not flg1))
          (app (det (listn nq2 'q) oracle3)
               (app (listn s2 flg1)
                    rest))))
```

We want to prove that scanning *waveform* past **flg1** and then **cdr**ing 10 times moves us into the sweet spot, which is underlined above.

But **scan** distributes over a series of ramps to produce

```
(cdrn 10
      [app (det (listn (no flg1 nq1 oracle1) 'q)
                (scan-oracle flg1 nq1 oracle1))
           (app (listn s1 (b-not flg1))
```

```
                    (app (det (listn nq2 'q) oracle3)
                         (app (listn s2 flg1)
                              rest)))])
```

since we know *s1* is nonzero. Let **(no flg1** *nq1* **oracle1)** be abbreviated by *no*. Observe that 0≤*no*≤*nq1*≤3. We can drive the **cdrn** past the first three **app**s to get: **(app (listn** *s2'* **flg1)** *rest***)** where *s2'* is the nonzero quantity *no* + *s1* + *nq2* + *s2*–10, provided we can show that

$$no+s1+nq2 \leq 10 < no+s1+nq2+s2.$$

The two inequalities above are necessary and sufficient for the **cdrn** to go past the first three **app**s and to stop without going past the fourth one. It remains therefore only to prove these two inequalities.

We do this by considering the known bounds on each term. See Figure 13. The terms *no*, *s1* and *nq2* are bounded above by 3, 4 and 3 respectively. The sum of the upper bounds is thus 3+4+3 = 10 and since 10 ≤ 10 we have established the first of our two inequalities.

The lower bound on *no* is 0. The lower bounds on *s1* and *nq2* are 1 and 1 respectively. The lower bound on *s2* is 9. The sum of the lower bounds is thus 0+1+1+9 = 11 and since 10 < 11, we have established the second inequality.

This completes the proof of base case 1 when **b** is **t**. Since the sweet spot is even bigger when **b** is **f**, the proof of that case is trivial and omitted.

Induction Case: We move on now to the induction case. The case is defined by the condition that **(listp msg)** and **(listp (cdr msg))** are both true. We inductively assume an instance of BPM18-Lemma in which **msg** is replaced by **(cdr msg)** and certain substitutions are made for the variables **flg1**, **nq**, **oracle1**, **dw**, **flg2**, **ts**, **tr**, and **oracle2**. We will derive the instance and exhibit it during the course of the proof. The induction conclusion is, of course, BPM18-Lemma itself.

Consider, again, the left-hand side, *lhs*,

```
(recv (len msg) flg1 10
      (app (det (listn nq 'q) oracle1)
           (det
            (warp
             (cdrn dw
                   (smooth flg2
                           (app (cdr (cells flg1 5 13 msg))
                                (listn p2 t))))
             ts tr w r)
            oracle2))).
```

Since **msg** is non-**nil**, the first argument, **(len msg)**, is equal to **(add1 (len (cdr msg)))**. In addition, using the same series of distributivity laws used in base case 1, we can simplify the other underlined terms above so as to reduce the last argument to the following, which we abbreviate as

waveform:

```
(app (det (listn nq 'q) oracle1)
     [app (det
           (warp
            (smooth flg2
                    (cell flg1
                          (difference 4 dw)
                          13
                          (car msg)))
            ts tr w r)
```

```
          oracle2)
         rest]))),
```

where

```
rest:
(app (det (listn (nq (difference 17 dw) ts tr w r)
                 'q)
         oracle3)
     (det
      (warp
       (cdrn (dw (difference 17 dw) ts tr w r)
             (smooth (b-not (csig flg1 (car msg)))
                     (app (cdr (cells (csig flg1 (car msg))
                                      5 13
                                      (cdr msg)))
                          (listn p2 t))))
       (ts (difference 17 dw) ts tr w r)
       (tr (difference 17 dw) ts tr w r)
       w r)
      oracle4))
```

and *oracle3* and *oracle4* are `oracle*` expressions whose values are unimportant.

At this point, *lhs* is equal to `(recv (add1 (len (cdr msg))) flg1 10` *waveform*`)`. By opening `recv` we reduce *lhs* to

```
[cons (recvbit 10 (scan flg1 waveform))
      (recv (len (cdr msg))
            (car (cdrn 10 (scan flg1 waveform)))
            10
            (cdrn 10 (scan flg1 waveform)))].
```

Following the analysis we did in base case 1, we know that the `recv-bit` expression is equal to `(car msg)`. Similarly, we reduce the `cdrn`/`scan` expression to `(app (listn` *s2'* `(csig flg1 (car msg)))` *rest*`)`.

Thus, *lhs* is

```
(cons [car msg]
      (recv (len (cdr msg))
            (car [app (listn s2' (csig flg1 (car msg))) rest])
            10
            [app (listn s2' (csig flg1 (car msg))) rest])).
```

But since *s2'* is nonzero, the `car` expression above is just `(csig flg1 (car msg))`. Using the fact that `recv` scans past leading occurrences of this signal in the `app` we get that *lhs* is

```
(cons (car msg)
      (recv (len (cdr msg))
            [csig flg1 (car msg)]
            10
            rest)).
```

We choose for our induction hypothesis that instance of BPM18-Lemma that establishes that the `recv` term above is equal to `(cdr msg)`. Suppose we can do that and relieve the hypotheses of the induction hypothesis. Then *lhs* is `(cons (car msg) (cdr msg))` which is `msg` and we are done.

It remains only to demonstrate that the **`recv`** term above is in fact an instance of the left-hand side of the conclusion of BPM18-Lemma (and to show that the instantiation chosen satisfies the hypotheses of BPM18-Lemma). Replacing the abbreviation *rest* by its meaning we see that the **`recv`** term is

```
(recv (len (cdr msg))
      (csig flg1 (car msg))
      10
      (app (det (listn (nq (difference 17 dw) ts tr w r)
                       'q)
                 oracle3)
           (det
            (warp
             (cdrn (dw (difference 17 dw) ts tr w r)
                   (smooth (b-not (csig flg1 (car msg)))
                           (app (cdr (cells (csig flg1 (car msg))
                                            5 13
                                            (cdr msg)))
                                (listn p2 t))))
             (ts (difference 17 dw) ts tr w r)
             (tr (difference 17 dw) ts tr w r)
             w r)
            oracle4))).
```

To obtain this term from the left-hand side of the conclusion of BPM18-Lemma, the following replacements suffice.

```
msg          (cdr msg)
flg1         (csig flg1 (car msg))
nq           (nq (difference 17 dw) ts tr w r)
oracle1      oracle3
dw           (dw (difference 17 dw) ts tr w r)
flg2         (b-not (csig flg1 (car msg)))
ts           (ts (difference 17 dw) ts tr w r)
tr           (tr (difference 17 dw) ts tr w r)
oracle2      oracle4
```

Inspection will also show that these instantiations satisfy the hypotheses of BPM18-Lemma, i.e., the new value of **`msg`** is a bit vector, the new value of **`nq`** is no greater than 3, the new value of **`dw`** is 0 or 1, the new values of **`ts`** and **`tr`** are **`clock-params`** with **`w`** and **`r`**, and the new flags are Boolean and opposite. Thus, the induction hypothesis establishes that the **`recv`** term above is indeed equal to **`(cdr msg)`**. **Q.E.D.**

The proof described here is essentially that checked by NQTHM. The complete transcript of the session in which NQTHM is led from its **`GROUND-ZERO`** theory to BPM18 is available on request from the author. The transcript contains 53 definitions and 208 theorems stated by the user so as to lead NQTHM to the proof. Roughly half of those theorems are elementary properties of natural number arithmetic and list processing functions such as **`app`** and **`len`**. The careful reader of the transcript will note that the mechanically checked proof differs tactically from the one described here primarily in elevating the case splits on the message bit; in our description of the proof we delayed those splits to conserve effort, while NQTHM tends to split early to simplify cases locally at the expense of duplicating work later. The total time required by NQTHM to process all of the definitions and theorems is about one hour on a Sun Microsystems 3/60.

8. Other Configurations of Biphase Mark

Most of the proof above concerned straightforward applications of our theory of **async** to the biphase mark output. The crucial step was the derivation of the inequalities

$$no+s1+nq2 \leq 10 < no+s1+nq2+s2$$

in base case 1. It should be clear that the numbers 5, 13, and 10 for the subcell sizes and sampling distance were chosen precisely to satisfy these two inequalities while reducing the cell size and the sampling distance. If we implemented **send** and **recv** with microprocessors nominally clocked at 20MHz each, then at 18 cycles per bit, the protocol would permit messages to be communicated at the burst rate of 1.1M bps. But note that we achieved 18 cycles per bit by an asymmetric division of the bit cell; our mark subcell is only 5 cycles long and hence our protocol demands a higher frequency response from the wire than is evident from the fact that our cell size is 18. By reducing the sampling distance we increase the protocol's tolerance for clock rate disparity.

An analogous proof can be constructed for other values of these parameters, provided the basic inequalities hold. In particular, if cell size 32 is chosen, with mark and code subcells of equal length and sampling distance 23, and we modify **rate-proximity** to give us $\frac{31}{32} \leq \frac{\mathtt{w}}{\mathtt{r}} \leq \frac{33}{32}$, the analogous inequalities are $3+15+3 \leq 23 < 0+12+1+12$. Because these inequalities hold, we see that the 32-cycle symmetric biphase mark protocol always recovers the bit correctly, provided the ratio of the clock rates are within $\frac{1}{32}$ (or 3.125%) of unity. From this remark it should be clear that we could undertake the proof of a more general theorem in which variables replace the particular subcell sizes and sampling distance and the clocks are constrained in relation to those variables. We have not undertaken the proof of that more general theorem because our main interest here was demonstrating that one particular version of the protocol works.

An interesting configuration to consider is cell size 16, split symmetrically into mark and code subcells, with sampling distance 11. The analogous inequalities are $3+7+3 \leq 12 < 0+4+1+4$, which are invalid. That is, the proof breaks down for the 16-cycle symmetric biphase mark protocol. This is not to say that the 16-cycle version does not work! Such a configuration is used in the Intel 82530 Serial Communications Controller [17] (where it presumably works) and we have found no example of reasonably close clock rates for which it fails in our model. But we cannot prove that it works using the attack shown here. Our attack bounds a sum by summing the bounds, which gives sound but crude results. The 16-cycle version, if indeed it works under our model, will require a more careful analysis of the bounds. It is also possible that the 16-cycle version is not correct under our model but that it works in practice. If this is the case, it it just illustrates the conservative nature of our model.

While the theorem establishes that the 18-cycle protocol works provided the clocks are within about 5%, experiments with the formal model suggest that the clock rate restriction can be considerably relaxed. We conjecture the 18-cycle protocol works for clock rate ratios that vary almost 30% from unity. Experiments show that the first place that the protocol fails to recover the first bit as the receiver's clock slows down in steps of 1 from the writer's clock of 100 is when the receiver's clock is 143. In particular,

```
(recv 4 t 10
      (async (send (list t f t t) 10 5 13 10)
             0 84 100 143
             (list t t f f)))
```

is **(list f t t t)**.

Thus, we believe the theorem we have proved about the 18-cycle protocol is very weak compared to what is true in the model. The culprit is our casual treatment of the bounds.

Our primary interest in this paper is not establishing the performance bounds of biphase mark. It is in explicating our model, demonstrating that it can be used to derive performance bound, and appealing to the engineering community to criticize its accuracy. Only after the model has survived the initial scrutiny of the engineering community do we feel it worthwhile to use it in a detailed formal study of communications.

9. Concluding Remarks on our Model

We have formalized a model of asynchrony that permits quantitative formal analysis of performance. We have taken a step toward developing a body of theorems about the model to permit its economical application to diverse problems.

We used the model to show that two different versions of the biphase mark procotol ‘‘work.’’ In the first protocol we send each bit in a cell lasting 18 cycles, the first 5 of which constitute the marking edge of the cell. We prove that the protocol permits the correct transmission of messages of arbitrary length provided the ratio of the clock rates of the two processors is within about 5% ($\frac{1}{18}$) of unity. The 18-cycle protocol gives a burst bit rate of about 1.1M bps if the processors have 20MHz clocks—though pin limitations on the actual implementation of the communication modules would require quantizing long messages and would degrade sustained performance. Furthermore, our 18-cycle protocol demands higher frequency response of the wire than is evident because the mark subcell is only 5 cycles long. We offer the 18-cycle protocol primarily as a catalyst for thought: The model says it will work. Will it?

We also used the model to show that the conventional 32-cycle biphase mark protocol allows correct transmission provided the clock rate ratio is within 3.125% of unity.

All of the proofs described here were checked with NQTHM. Inevitably, the reader of this paper will wonder if there are mistakes in our presentation of the proof. Indeed, so does the author. Does each formula follow from the previous one? While these doubts inevitably arise in the context of a proof presented on paper, they do not arise during the machine-assisted act of creating the proof in the first place. Furthermore, the user of NQTHM is concerned primarily with inventing the lemmas that enable the rewrite steps and not with the construction or even the derivation of the terms that thereby arise. One of the main advantages to having a formal model in a mechanized logic is that it is possible to have machine assistance while exploring the ramifications of various decisions.

Returning to our model *per se*, it is presented as a recursively defined function on waveforms. To use it to investigate the communication from one processor to another it is (only) necessary to formalize the input/output behavior of the two processors. The implementation details of each processor are not relevant. Furthermore, each processor may be specified independently of the other.

Because of this decomposition, it is possible to verify an implementation of each processor independently of the other and of the model of asynchrony. Consider **send**. It is the formal specification of the kernel of the send side of a microprocessor’s communications module. Indeed, its definition was developed with that use in mind. See [25]. Using the Formal HDL described in [7], it is possible to design a circuit that implements **send**. The formal semantics of the HDL is cast as an NQTHM interpreter (or simulator) that determines the signals on all the pins and the state produced by a described design, given the initial signals and state. Thus one can easily define the sequence of signals produced by a circuit. Suppose we had a circuit alleged to implement **send**. That means the sequence of signals on a given pin over some number of cycles starting from a given initial state is equal to the sequence of signals produced by **send**. Proving such a correctness result would be straightforward (given the reusable theory developed for the Formal HDL by Brock and Hunt) for some hardware designs. See [25] for an example of the use of the Formal

HDL in the specification and design of a simple verified microprocessor.

In an exactly analogous fashion, one could design a digital phase locked loop alleged to implement **`recv`** and prove that it was correct.

Our point about decomposition is that the proofs of correctness of these two hardware modules are independent, both of eachother and of our model of asynchrony. The Formal HDL provides the ability to verify synchronous designs (designs in which there is only one clock) and that is all we need to design and verify implementations of **`send`** and **`recv`**. Given two verified processors one can then establish that they communicate properly by applying our model and reasoning about their specifications rather than their implementations. That is what we have done in this paper: we proved that **`send`** and **`recv`** — the specifications of two independently clocked synchronous processors — provide reliable communication.

A limitation of our model is that it only addresses one-way communication. There is no way to use it to verify two-way communication if timing or ordering on the signals is relevant (as it is in true two-way communication). This is a general problem that has nothing to do with asynchrony but rather with message passing formalized at the level of independently specified input/output streams. Perhaps the general problem can be solved in a way that delays consideration of the effects of asynchrony and transforms the dialog into two monologues (having certain oracular properties that permit their interpretation as a dialog) that can then be investigated by the techniques developed here. In any case, we see this as a fruitful area of further research.

Another limitation of our model is that we have assumed that clocks are linear functions of time. We do not know how inaccurate this assumption is. A more general model is that clocks are nearly linear in the sense that every cycle is within some epsilon of the nominal length. This could be formalized in the style given here. There is no doubt that it would complicate the reusable theory of **`async`**. Determining the lengths of the various regions of the warped signal would be more tedious. We speculate that the accumulating clock error would tend to be washed out by our conservative treatment of edges and would not be fatal to the proof of the biphase mark protocol.

Finally, our model ignores various engineering realities such as metastability, reflections, noise, and distortion. It was our intention to ignore these on the grounds that we wanted to address the problems of asynchrony rather than of signal processing. This attempt to separate concerns may be misguided: some protocols are designed to overcome noise, say, and the entire *raison d'etre* of such designs is lost in our analysis.

In the end we must come back to our introductory remarks on engineering. We have formalized a model of asynchrony. With the model it is possible to prove that certain protocols work. It is up to the engineer to decide whether the model is accurate enough for the purposes at hand.

10. Relation to Other Work

This work began as part of a NASA-sponsored investigation at Computational Logic, Inc. (CLI) into the formalization of fault tolerance. W. Bevier and W. Young of CLI formalized with NQTHM the Oral Messages (or ''Byzantine Agreement'') algorithm of Pease, Shostak, and Lamport [26]. In [4] they describe the formalization and correctness proof of that algorithm and carried it all the way down to the NQTHM specification of four microprocessors that use the algorithm to reach agreement in the presence of faults. Young then used NQTHM to prove the correctness of the interactive convergence clock synchronization algorithm, essentially following in the footsteps of Rushby and von Henke [29]. Meanwhile, the present author used the hardware description language formalized in NQTHM by B. Brock and W. Hunt [7] of CLI to implement the processor specified by Bevier and Young and to prove that the described design meets their specification [25]. The clear but unstated direction of the CLI work on fault-tolerance was to enable the eventual fabrication of a device implementing the Byzantine agreement algorithm—a device whose design had been mechanically verified from the journal article describing the algorithm all the way down to the netlist. (See [3] for a description of the similarly verified ‘‘CLI short stack’’ that goes from a verified compiler for a simple high-level language, through a verified assembler and linker, to a microprocessor verfied at the gate level.) However, a major stumbling block in this program was the fact that the four microprocessors specified by Bevier and Young were unrealistically assumed to execute in lockstep synchrony, i.e., to share a common clock. This is unacceptable since it introduces a potential single-point failure into the system. This assumption was made primarily to enable the convenient exchange of data between the four processors during the voting that leads to agreement. It was therefore natural to study the question of verified communication between asynchronous processors. It should be noted that even with all the present pieces in place, the goal of a verified network of asychronous Byzantine processors is still a significant challenge.

Our model of asynchronous communication is expressed as a function that transforms the signal stream produced by one processor into the signal stream consumed by an asynchronous processor. To apply the model, one must characterize the signals produced and consumed by the two communicating processes. This input/output model of concurrent processes is a familiar one used in Milner’s CCS [23] and Hoare’s CSP [15]. Unlike that work, we consider only the simple case of one way communication. However, our focus is entirely on the physical problems introduced by asynchrony, namely how clock rates, delay, and phase shift affect the received signal. The quantitative modeling of time makes our work very different in character and focus from the cited work. The reader interested in the general problems of verifying distributed and/or concurrent systems should see, in addition to [23] and [15], the seminal work by Manna and Pnueli [21], Barringer’s survey [2], and the Unity model by Chandy and Misra [9]. In [14], D. Goldschlag describes an NQTHM-based mechanized proof system Unity.

Our work finds its closest relatives in the very active field of hardware verification. See [33] for a tutorial introduction to and overview of the field. In common with our work, many formal models of microprocessors, e.g., [16], [32], and [20], quantitatively measure time in cycles. A particularly intriguing title, given the title of this work, is J. Joyce’s ‘‘Formal Specification and Verification of Asynchronous Processes in Higher-Order Logic’’ [19]. The report deals with the same problem confronted in [16], namely how to formalize the interaction between a synchronous microprocessor and an asynchronous memory via a four-phase handshaking protocol. The report offers an attractive alternative to the formalization presented in [16]. But it does not address general asynchronous communication in the sense that we do.

Because we verify a protocol in this paper, it is necessary to comment upon the relation of our work to the very old and very active research area of protocol verification. An important survey of the field was published as long ago as 1979 [31] and the field has an annual conference (Protocol Testing, Specification, and Verification) with proceedings published by North-Holland [1].

The International Standards Organization has defined seven levels of protocol. Level 1, the ‘‘physical level,’’ deals with pin connections, voltage levels, and physical signal formats. Level 2, the ‘‘data link level,’’ concerns itself with data formats, synchronization, error control, and flow control. Above those are, successively, the ‘‘network level,’’ the ‘‘transport level,’’ the ‘‘session level,’’ the ‘‘presentation level’’ and the ‘‘application level.’’

Perhaps the most easily distinguished feature of our work is that it is essentially at level 1 while, to the best of our knowledge, all other formal verification work on protocols addresses higher levels.

The best studied protocol is probably the alternating bit protocol, which is at level 2. Of special concern in that protocol is detection of message loss to an unreliable lower level. The protocol provides for acknowledgement of reception (which may itself get lost) and retransmission (which may lead to duplicate receptions). In the late 70s mechanical protocol verification was based on the then-standard program verification technology: a procedural encoding of the protocol was annotated with inductive assertions, from which verification conditions were mechanically generated and then interactively proved. In [12] this method is applied to the alternating bit protocol. See [13] for examples of method applied to still-higher transport level protocols. But in the 80s the combination of finite-state machine models, propositional temporal logic, and fast mechanical decision procedures came to dominate mechanized protocol verification because of the speed and automation this combination offered. For a description how this approach is applied to the alternating bit protocol see [10] by E. Clarke, E. Emerson and A. Sistla. Clarke and O. Grumberg have written an excellent review of the use of finite state machines and temporal logic in automatic verification of concurrent systems [11].

However, the finite state machine approach and the related Petri net approach [27] suffer from the inability to discuss time quantitatively. Much research in the protocol verification community is now aimed at adding some notion of time to the finite state approach, without exacerbating the already vexing state explosion problem or taking the entire problem out of the propositional domain. This is in stark contrast to our work, where explicit, quantitatively measured time forms the foundation of the model.

Finally, while not at level 1 and not supported by mechanically checked proofs, the closest work on protocol verification is perhaps that by P. Jain and S. Lam [18] where time is modeled quantitatively and discretely and signal propagation down a bus is also modeled (assuming constant propagation speed). They specify a modified Expressnet protocol which they prove to be collison-free and they derive bounds for its access delay.

11. Acknowledgements

I would like to thank my colleagues at Computational Logic, Inc., for helping me with this project, especially Bill Young and Bill Bevier for bringing the formalization of asynchrony to my attention, Warren Hunt for explaining microprocessors and biphase mark, Bishop Brock for explaining metastability and his and Warren’s formal HDL to me, Larry Smith for explaining how Bill and Bill’s Byzantine processor specification actually told him how to build it, and Matt Wilding for helping find algebraic expressions for certain of the functions in the model. None of these people should be held responsible for my misconceptions regarding hardware and communications.

References

1. S. Aggarwal and K. Sabnani (eds.). *Protocol Specification, Testing, and Verification VIII.* Elsevier Science Publishers B.V. (North-Holland), 1988.

2. H. Barringer. *A Survey of Verification Techniques for Parallel Programs.* Springer-Verlag Lecture Notes in Computer Science 191, Berlin, 1985.

3. W.R. Bevier and W.A. Hunt and J S. Moore and W.D. Young. "Special Issue on System Verification". *Journal of Automated Reasoning 5*, 4 (1989), 409-530.

4. W.R. Bevier and W.D. Young. The Proof of Correctness of a Fault-Tolerant Circuit Design. Proceedings of the Second International Working Conference on Dependable Computing for Critical Applications, February, 1991, pp. 107-114.

5. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

6. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.

7. B.C. Brock and W.A. Hunt. A Formal Introduction to a Simple HDL. In *Formal Methods for VLSI Design*, J. Staunstrup, Ed., Elsevier Science Publishers B.V. (North-Holland), 1990, pp. 285-329.

8. J. Campbell. *C Programmer's Guide to Serial Communications.* Howard W. Sams and Co., 4300 West 62 Street, Indianapolis, IN 46268, 1988.

9. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, Massachusetts, 1988.

10. E.M. Clarke and E.A. Emerson and A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems 8*, 2 (1986), 244-263.

11. E.M. Clarke and O. Grumberg. "Research on Automatic Verification of Finite-State Concurrent Systems". *Ann. Rev. Comput. Sci. 2* (1987), 269-290.

12. B.L. DiVito. A Mechanical Verification of the Alternating Bit Protocol. Tech. Rept. ICSCA-CMP-21, Institute for Computing Science, The University of Texas at Austin, 1981.

13. B.L. Di Vito. Verification of Communications Protcols and Abstract Process Models. PhD Thesis ICSCA-CMP-25, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703..

14. D.M. Goldschlag. Mechanizing Unity. In *Programming Concepts and Methods*, M. Broy and C. B. Jones, Eds., North Holland, Amsterdam, 1990.

15. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, Englewood Cliffs, NJ, 1985.

16. W.A. Hunt. FM8501: A Verified Microprocessor. University of Texas at Austin, December, 1985. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703..

17. Intel Corporation. *Microcommunications.* Intel Literature Sales, P.O. Box 7641, Mt. Prospect, IL 60056-7641, 1991.

18. P. Jain and S.S. Lam. "Specification Real-Time Protocols for Broadcast Networks". *IEEE Transactions on Computers 40*, 4 (1991), 404-422.

19. J.J. Joyce. Formal Specification and Verification of Asynchronous Processes in Higher-Order Logic. Tech. Rept. Technical Report No. 136, University of Cambridge Computer Laboratory, June, 1988.

20. J.J. Joyce. Multi-Level Verification of Microprocessor-Based Systems. Tech. Rept. Technical Report No. 195, University of Cambridge Computer Laboratory, May, 1990.

21. Z. Manna and A. Pnueli. "Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs". *Science of Computer Programming 4* (1984), 257-289.

22. C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley Publishing Co., 1980.

23. R. Milner. *A Calculus of Communicating Systems.* Springler-Verlag, Berlin, 1980.

24. F.C. Mish (Ed.) *Webster's Ninth New Collegiate Dictionary.* Merriam-Webster, Inc, 1987.

25. J S. Moore. Mechanically Verified Hardware Implementing an 8-Bit Parallel IO Byzantine Agreement Processor. Tech. Rept. NASA CR-189588, NASA, 1992.

26. M. Pease and R. Shostak and L. Lamport. "Reaching Agreement in the Presence of Faults". *Journal of the ACM 27*, 2 (1980), 228-234.

27. J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.

28. M. S. Roden. *Digital Communication Systems Design.* Prentice Hall, 1988.

29. J. Rushby and F. von Henke. Formal Verification of the Interactive Convergence Clock Synchronization Algorithm using EHDM. Tech. Rept. SRI CSL 89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, January, 1989.

30. B. Sklar. *Digital Communications Fundamentals and Applications.* Prentice Hall, 1988.

31. C. Sunshine. "Formal Techniques for Protocol Specification and Verification". *Computer 12*, 9 (1979), 20-27.

32. C.H. Pygott. Formal Proof of Correspondence Between the Specification of a Hardware Module and its Gate Level Implementation. Report 85012, Royal Signals and Radar Establishment, Malvern, Worcestershire (United Kingdom), November, 1985.

33. M. Yoeli. *Formal Verification of Hardware Design.* IEEE Computer Society Press, Los Alamitos, California, 1990.

Table of Contents

List of Figures