Formal Aspects of Computing (1993) 3: 1–000
© 1993 BCS

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

J Strother Moore

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Tx 78703-4776 U.S.A.

Abstract. We present a formal model of asynchronous communication between two digital hardware devices. The model takes the form of a function in the Boyer-Moore logic. The function transforms the signal stream generated by one processor into that consumed by an independently clocked processor, given the phases and rates of the two clocks and the communications delay. The model can be used quantitatively to derive concrete performance bounds on communications at ISO protocol level 1 (physical level). We use the model to show that an 18-bit/cell biphase mark protocol reliably sends messages of arbitrary length between two processors provided the ratio of the clock rates is within 5% of unity.

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

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

Correspondence and offprint requests to: J Strother Moore, Computational Logic, Inc., 1717 West Sixth Street, Suite 290, Austin, Texas 78703-4776, USA. Email: moore@cli.com

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) sketch 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. The proof was checked with the Boyer-Moore theorem prover, Nqthm [BM88].

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[Int91]. A version of biphase mark, called "Manchester," is used in the Ethernet[Rod88] and is implemented in the Intel 82C501AD Ethernet Serial Interface[Int91]. Biphase mark is also used in some optical communications and satellite telemetry applications[Skl88]. 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. 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.

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

[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."

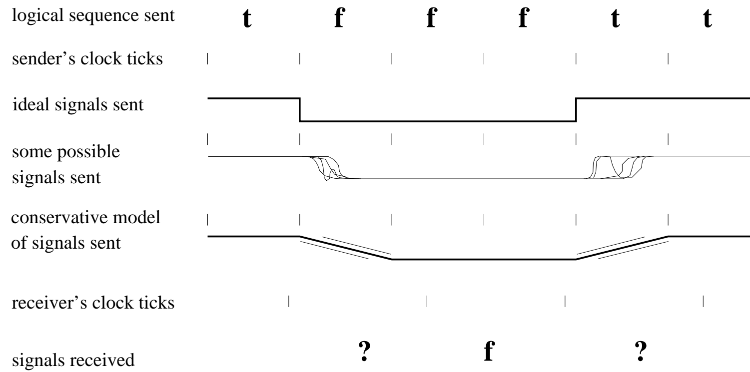

Fig. 1. How Asynchrony Mangles Signals

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 full cycle, that cycle spanning the first ramp. If the receiver samples the signal stream during this cycle the logical result obtained depends upon exactly when during the cycle the signal enters into the combinational logic of the receiver. It may happen early in the cycle (in which case the receiver may get a **t** because the signal is high) or late in the cycle (producing **f**) or in the middle (in which case the signal may be interpreted either way or may even induce a "metastable" state in which the whole notion of its "logical value" is ill-defined and the receive ceases, perhaps temporarily, to act as a digital device). Furthermore, the exact time at which the signal is sensed during a cycle may vary from cycle to cycle in a data dependent way. At this stage in our model, therefore, we simply say the signal received during this cycle is undetermined.

Things are simpler during the receiver's second full cycle; the signal is constant at **f** for the duration of the cycle and hence we are assured that it reads an **f** no matter when during the cycle the line is sensed.

The problem of metastability caused by "reading on an edge" cannot be solved perfectly by digital logic alone. 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.

[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.

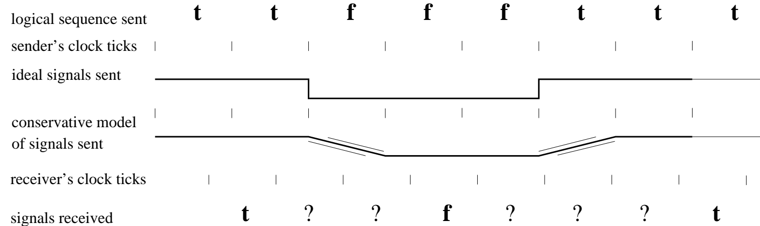

Fig. 2. One Edge Can Influence Several Read Cycles

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, 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,

[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[Cam88]. 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"[Mis87]. Thus we speak of the "mark subcell," so named because it indicates the beginning of the cell, and of "detecting the mark."

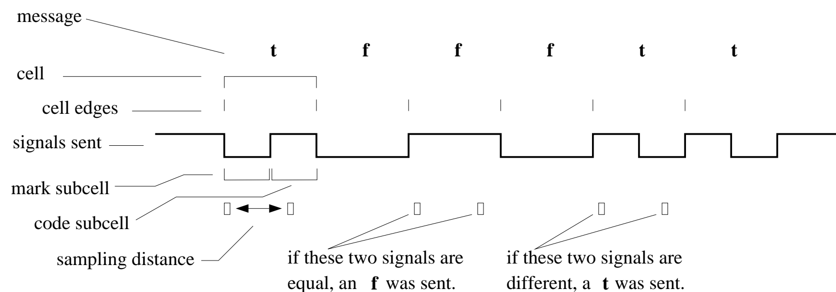

Fig. 3. Biphase Mark Terminology

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 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 define a model of asynchrony and we formally

state 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 [RvH89] 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.

This article is a shortened version of [Moo92a], where we present the proof in full as well as develop a "reusable theory" that allows the application of our model to other protocols.

2. Logical Foundations

We use the Nqthm "computational logic" described in [BM88].

Truth values, bits, and signals will all be represented by the objects **t** and **f** which are distinct constants. We call these two objects "Booleans." Because the logic's language is untyped, we define a predicate, boolp, which recognizes just them.

DEFINITION: *boolp*
boolp (x)
=
$((x = \mathbf{t}) \vee (x = \mathbf{f}))$

As can be seen 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[BM79, BM88] 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.

DEFINITION: *b-not*
b-not (x)
=
$(\neg\ x)$

DEFINITION: *b-xor*
b-xor $(x,\ y)$
=
if x
then $\neg\ y$
elseif y
then t
else f
endif

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 to represent such objects. The following function recognizes bit vectors.

DEFINITION: *bvp*
bvp (x)
=
if $x \simeq$ **nil**
then $x =$ **nil**
else boolp (car (x))
$\wedge$
bvp (cdr (x))
endif

That is, bvp (x) is defined by cases. If x is an atom[4] then x is a bit-vector precisely when x is **nil**. On the other hand, 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. 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 nth element of the list x (where car (x) is the 0th 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 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

[4] In the Nqthm logic, $x \simeq$**nil** actually means x is not a listp object. Any such "atom" may be used to end a list, but **nil** is the most common.

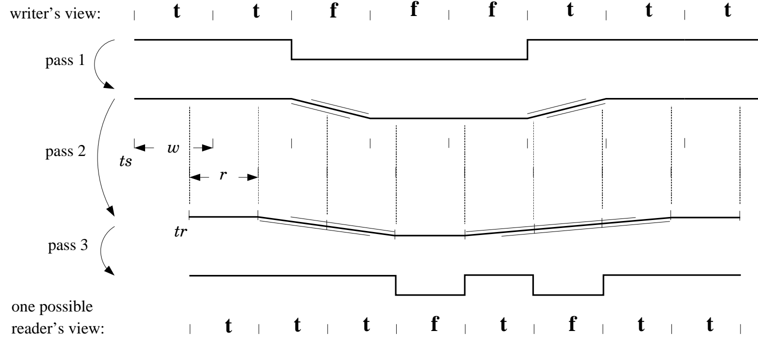

Fig. 4. The Three Passes in the Model

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[Rod88].
- 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.
- 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 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 signal stream. 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 exactly 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 object `'q`. A `'q` at a given cycle in the stream signifies that the voltage on the line is indeterminate during that cycle. It is misleading to think of `'q` either as a signal that can be recognized by the receiver or as a wildcard that nondeterministically denotes a `t` or `f`. It is an artifact of the model signifying "here the signal is unknown." The model can recognize the presence of indeterminacy and even propagate it. Ultimately, the model replaces `'q`'s nondeterministically by the Booleans sensed by the receiver. 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*
smooth (x, *lst*)
  =
if *lst* $\simeq$ **nil**
  then nil
elseif b-xor (x, car (*lst*))
  then cons (`'q`,
          smooth (car (*lst*), cdr (*lst*)))
else cons (car (*lst*),
        smooth (car (*lst*), cdr (*lst*)))
endif

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**, $\underline{\mathbf{f}}$, **f**, **f**, $\underline{\mathbf{t}}$, **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.[5] 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 $ts \leq tr < ts+w$ because if $tr \geq ts+w$ 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*
warp (lst, ts, tr, w, r)
  =
if ($r \simeq 0$)
  $\vee$
  endp (lst, ts, $tr + r$, w)
 then nil
else cons (sig (lst, ts, $tr + r$, w),
  warp (lst' (lst, ts, $tr + r$, w),
   ts' (lst, ts, $tr + r$, w),
   $tr + r$,
   w,
   r))
endif

Observe that $tr+r$ is the time at which the reader's clock next ticks. The definition may be read as follows: If r is zero[6] 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 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.

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, $tr + r$, 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, $tr + r$, w),
 ts' (lst, ts, $tr + r$, w),

[5] 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 $\omega + \delta$. We assume δ is constant.
[6] Actually, $r \simeq 0$ here is **t** if and only if r is either not a natural number or is 0. Omitting this test produces an inadmissible definition because the recursion described does not terminate.

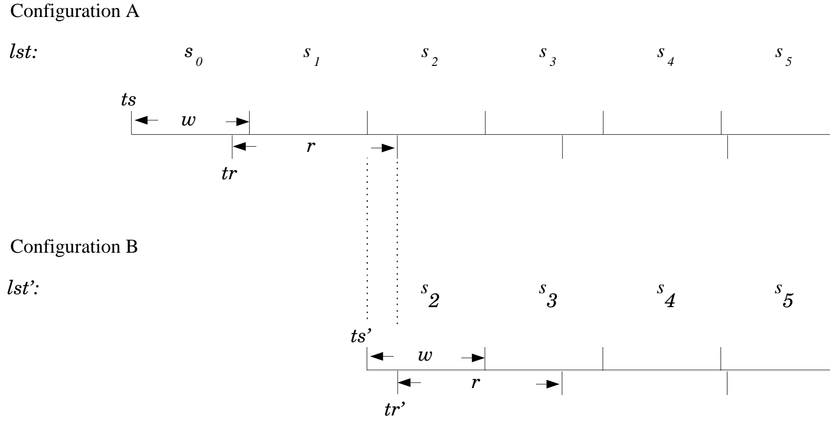

Fig. 5. The Recursion in warp

$tr + r$,
w,
r)

The easiest argument term to understand is 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 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*
endp (*lst*, *ts*, *tr*, *w*)
=
if *lst* $\simeq$ **nil**
then t
elseif $(ts + w) < tr$

```
  then endp (cdr (lst), ts + w, tr, w)
else f
endif
```

DEFINITION: *reconcile-signals*

```
reconcile-signals (a, b)
  =
if a = b
  then a
else 'q
endif
```

DEFINITION: *sig*

```
sig (lst, ts, tr, w)
  =
if lst ≃ nil
  then 'q
elseif (ts + w) < tr
  then reconcile-signals (car (lst),
                          sig (cdr (lst), ts + w, tr, w))
else car (lst)
endif
```

DEFINITION: *lst'*.

```
lst' (lst, ts, nxtr, w)
  =
if lst ≃ nil
  then lst
elseif nxtr < (ts + w)
  then lst
else lst' (cdr (lst), ts + w, nxtr, w)
endif
```

DEFINITION: *ts'*

```
ts' (lst, ts, nxtr, w)
  =
if lst ≃ nil
  then ts
elseif nxtr < (ts + w)
  then ts
else ts' (cdr (lst), ts + w, nxtr, w)
endif
```

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[RvH89]. 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
det (lst, oracle)
  =
if lst ≃ nil
  then lst
elseif car (lst) = 'q
  then cons (if car (oracle)
                  then t
                else f
                endif,
                det (cdr (lst), cdr (oracle)))
else cons (car (lst),
              det (cdr (lst), oracle))
endif
```

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`

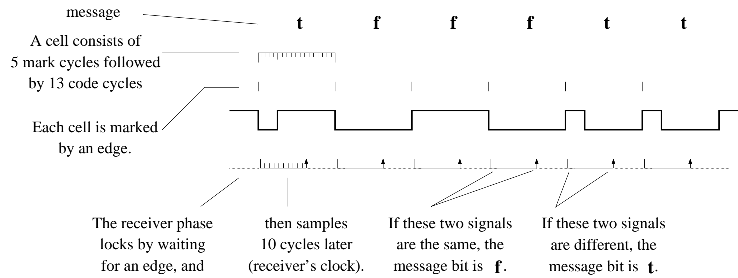

Fig. 6. Our Modified Biphase Mark Protocol

should be replaced by **t** or by **f**.[7] 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*
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.

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.

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

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 *csig* be x if b is **t** and b-not (x) otherwise. Then a cell is defined as the concatenation of a "mark" subcell containing n repetitions of b-not (x), followed by a "code" subcell containing k repetitions of *csig*.

```
DEFINITION: cell
cell (x, n, k, bit)
  =
app (listn (n, b-not (x)),
      listn (k, csig (x, bit)))
```

where

```
DEFINITION: csig
csig (x, b)
  =
if b
  then x
else b-not (x)
endif
```

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
cells (x, n, k, msg)
  =
if msg ≃ nil
  then nil
else app (cell (x, n, k, car (msg)),
          cells (csig (x, car (msg)),
                 n,
                 k,
                 cdr (msg)))
endif
```

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

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**)

Fig. 7. Sending list (**t**, **f**, **t**, **t**) with Cells of Size 1+2

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.

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 xs, scan finds the first edge.

DEFINITION: *scan*
scan (x, *lst*)
=
if *lst* $\simeq$ **nil**
then nil
elseif b-xor (x, car (*lst*))
then *lst*
else scan (x, cdr (*lst*))
endif

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*
recv-bit (k, *lst*)
=
if b-xor (car (*lst*), nth (k, *lst*))
then t
else f
endif

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

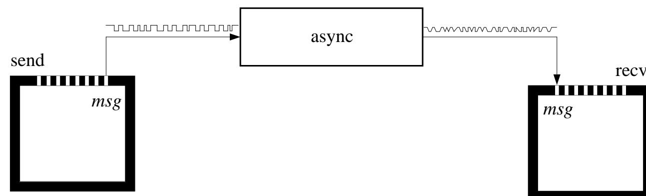

Fig. 8. The Composition of send, async and recv

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 xs 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*
recv (i, x, k, *lst*)
  =
if $i \simeq 0$
  then nil
else cons (recv-bit (k, scan (x, *lst*)),
    recv ($i-1$,
      nth (k, scan (x, *lst*)),
      k,
      cdrn (k, scan (x, *lst*))))
endif

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.

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

$$\text{recv}(\text{len}(msg),\ \mathbf{t},\ 10,\ \text{async}(\text{send}(msg,\ p1,\ 5,\ 13,\ p2),\ ts,\ tr,\ w,\ r,\ oracle))$$

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 $ts \leq tr < ts+w$. 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 $17/18 \leq w/r \leq 19/18$. But since we have limited ourselves to natural arithmetic, we define rate-proximity equivalently via

DEFINITION: *rate-proximity*

$$\text{rate-proximity}(w,\ r) = (((18 * w) \not< (17 * r)) \wedge ((19 * r) \not< (18 * w)))$$

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 prove is

THEOREM: BPM18

$$((\text{bvp}(msg) \wedge (ts \in \omega) \wedge (tr \in \omega) \wedge (w \not\simeq 0) \wedge (r \not\simeq 0) \wedge (tr \not< ts)$$

$$
\begin{aligned}
&\quad\wedge \\
&(tr < (ts + w)) \\
&\quad\wedge \\
&\text{rate-proximity}(w,\ r) \\
&\quad\wedge \\
&(p1 \in \omega)) \\
&\quad\rightarrow \\
&(\text{recv}\,(\text{len}\,(msg), \\
&\qquad \mathbf{t}, \\
&\qquad 10, \\
&\qquad \text{async}\,(\text{send}\,(msg,\ p1,\ 5,\ 13,\ p2),\ ts,\ tr,\ w,\ r,\ oracle)) \\
&\quad = \\
&msg))
\end{aligned}
$$

"BPM18" stands for "Biphase Mark, 18-cycles/bit." 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 w/r to be within 1/18 of unity. For what it is worth, 1/18 is $0.0\overline{5}$, or somewhat more than 5%.

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.

7. Proofs

BPM18 can be proved by transforming it into a slightly different form and then appealing to a more general theorem which is proved by induction.

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 develop a body of lemmas about async and

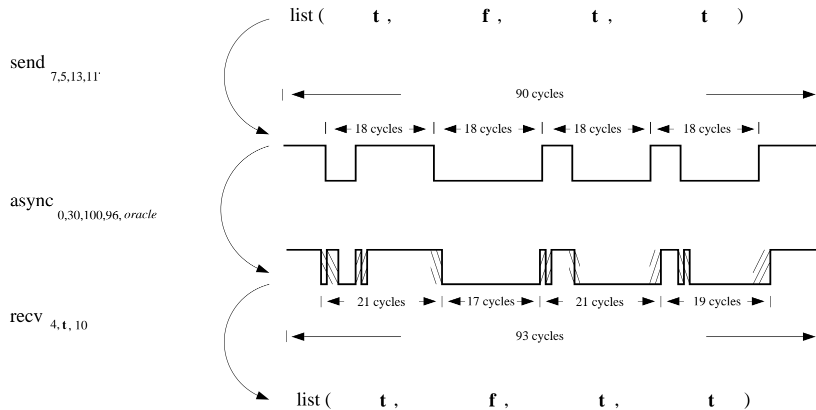

Fig. 9. An Experiment with send, async, and recv

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

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 listns to generate either "high" or "low" horizontal lines and then use apps 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 smooths. 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. See [Moo92a] for the presentation of the reusable theory.

7.2. A Sketch of the Proof of BPM18

THEOREM: BPM18

$$
\begin{array}{l}
((\text{bvp}\,(msg) \\
\quad\wedge \\
(ts \in \omega) \\
\quad\wedge \\
(tr \in \omega) \\
\quad\wedge \\
(w \not\simeq 0) \\
\quad\wedge \\
(r \not\simeq 0) \\
\quad\wedge \\
(tr \not< ts) \\
\quad\wedge \\
(tr < (ts + w)) \\
\quad\wedge \\
\text{rate-proximity}\,(w,\ r) \\
\quad\wedge \\
(p1 \in \omega)) \\
\quad\rightarrow \\
(\text{recv}\,(\text{len}\,(msg), \\
\qquad \mathbf{t}, \\
\qquad 10, \\
\qquad \text{async}\,(\text{send}\,(msg,\ p1,\ 5,\ 13,\ p2),\ ts,\ tr,\ w,\ r,\ oracle)) \\
\quad = \\
msg))
\end{array}
$$

We do not give the proof of BPM18 but merely sketch it to illustrate how the model is manipulated in the reusable theory. Complete details are given in [Moo92a].

We transform the left-hand side of the conclusion above into a slightly dif-

ferent form and then appeal to a lemma (discussed below). 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)).
```

Consider the second app term. We know its first 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)).
```

Using the reusable theory, we distribute the smooth, warp, and det down the waveform to obtain:

```
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))))).
```

Observe that the reusable theory introduces some new function symbols, e.g., n*, nq, and dw. These functions define the lengths of various portions of the transformed waveform. For example, the list of *p1* **t**s output by send is transformed into a region of n* (*p1*, *ts*, *tr*, *w*, *r*) **t**s by our model. The reusable theory tells us the length of this received region, expressed algebraically in terms of *p1* and the clock parameters.

Consider the first listn term above. It denotes 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 above manipulations are typical of those used in our proof. However, the proof was actually constructed with the assistance of a mechanical theorem prover. The user's role was "merely" to state the lemmas that drive the simplification above. The system proved the lemmas and then used them to carry out the manipulations. Put another way, the user's role was the extremely creative one of formalizing the reusable theory of async in such a way that it could be used to drive Nqthm to this proof. The system's role was the tedious application of this body of rules.

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

```
((bvp (msg)
      ∧
  (ts ∈ ω)
      ∧
  (tr ∈ ω)
      ∧
  (w ≄ 0)
      ∧
  (r ≄ 0)
      ∧
  (tr ≮ ts)
      ∧
  (tr < (ts + w))
      ∧
  rate-proximity (w, r)
      ∧
  (nq ∈ ω)
      ∧
  (3 ≮ nq)
      ∧
  (dw ∈ ω)
      ∧
  (1 ≮ dw)
      ∧
  boolp (flg1)
      ∧
  boolp (flg2)
      ∧
  b-xor (flg1, flg2))
     →
 (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

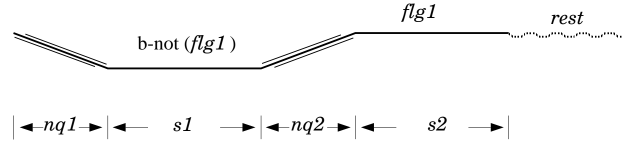

Fig. 10. *Waveform* if *flg1* is "High"

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*. See [Moo92a] for the details.

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 mark.

The reusable theory can be used to derive the general shape of the wave reaching the receiver. If the first bit of the message is **t** and *flg1* is depicted as "high" then the waveform has the shape shown in Figure 10.

The reusable theory also informs us as to upper and lower bounds on the lengths of each region of the received waveform. The bounds are shown in Figure 11. The constraints on the relative rates of the two clocks determine these bounds. 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.

The crucial step in proving BPM18-Lemma is showing that the sampling point (which is 10 reader cycles after the detection of the mark) is in the "sweet spot." This comes down to the following inequalities:

[8] We do not have time to expound upon the subtleties of the BPM18-Lemma formula, but the most creative part of the proof was finding a statement of this lemma that could be proved by induction—a statement with the property that an inductive instance could be used to prove the inductive conclusion. The difficulty here was not due to the limitations of the mechanical theorem prover we used but rather to the mathematical complexity of the situation. The trick was to find a general description of the input waveform at the receiver so that if that waveform is lengthened by one appropriately transformed cell and then the receiver processes the leading cell, the remaining signal stream to be processed is again in the general form specified.

$$1 \leq nq1 \leq 3$$
$$1 \leq s1 \leq 4$$
$$1 \leq nq2 \leq 3$$
$$9 \leq s2 \leq 12$$

Fig. 11. Bounds on the Lengths of Waveform Regions

$$no{+}s1{+}nq2 \leq 10 < no{+}s1{+}nq2{+}s2,$$

where $0 \leq no < 3$ is the number of cycles in the first ramp of the wave after the first cycle at which the nondeterministic read is low. By considering the known bounds on each of the terms (Figure 11), we see that $10 \leq 10 < 11$, and thus the sampling point is within the sweet spot.

The proof sketched 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 total time required by Nqthm to process all of the definitions and theorems is about twelve minutes on a Sun Microsystems SparcStation 2 GX-Plus, running Nqthm in AKCL. However, about a man-month was spent developing the reusable theory.

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 $31/32 \leq w/r \leq 33/32$, 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 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[Int91] (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% (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. It should be noted that it was not necessary to invent a specialized logic to reason effectively; indeed, facility in arithmetic is paramount here.

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.

Recall that our model does not specify when during a cycle the voltage on the pin is changed or sensed. This contributes to the pessimism of our models, since, for example, a read cycle that starts *before* an edge arrives can be "contaminated" by the edge because the model allows the possibility that the pin is sensed late in the cycle. The model further allows the pin to be read late in one cycle and early in another. The good effect of this permissivity is that to apply the model we do not have to delve into exactly what logic is used to sense the pin. Indeed, any two designs that implement the same function on the signal stream are equivalent.

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[Moo92b]. Using the Formal HDL described in[BH90], 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[Moo92b] 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. (Phase locking is the idea of adjusting the clocks of two or more processes so that all the clocks tick "simultaneously." A common technique is for the sender to encode its clock in the signal stream and for the receiver to adjust its timing accordingly. Phase locking is often done with special devices that change the rate at which crystals vibrate. But by adopting an artificially slow "virtual" clock, e.g., where one virtual tick occurs every n physical ticks, it is possible to implement phase locking in software or firmware. That is called "digital phase locking" and is increasingly popular. Biphase mark protocols are often used in such implementations.)

Our point about decomposition is that the proofs of correctness of these two hardware modules are independent, both of each other 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[PSL80]. In [BY91] 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[You91] then used Nqthm to prove the correctness of the interactive convergence clock synchronization algorithm, essentially following in the footsteps of Rushby and von Henke[RvH89]. Meanwhile, the present author used the hardware description language formalized in Nqthm by B. Brock and W. Hunt[BH90] of CLI to implement the processor specified by Bevier and Young and to prove that the described design meets their specification[Moo92b]. 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[BHMY89] 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[Mil80] and Hoare's CSP[Hoa85]. 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 [Mil80] and [Hoa85], the seminal work by Manna and Pnueli[MP84], Barringer's survey[Bar85], and the Unity model by Chandy and Misra[CM88]. In[Gol90], 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 [Yoe90] for a tutorial introduction to and overview of the field. In common with our work, many formal models of microprocessors, e.g., [Hun85], [Pyg85], and [Joy90], 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"[Joy88]. The report deals with the same problem confronted in [Hun85], 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 [Hun85]. 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[Sun79] and the field has an annual conference (Protocol Testing, Specification, and Verification) with proceedings published by North-Holland[Ae88].

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 [DiV81] this method is applied to the alternating bit protocol. See[DiV82] 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[CES86] 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[CG87].

However, both the finite state machine approach and the related Petri net approach[Pet81] 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[JL91] 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

This work was supported in part at Computational Logic, Inc., by the National Aeronautics and Space Administration, NASA Contract NAS1-18878. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., NASA or the U.S. Government.

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. Finally, I would like to thank the anonymous referee whose careful reading of the paper and helpful suggestions clarified some of the issues. None of these people should be held responsible for my misconceptions regarding hardware and communications.

References

[Ae88] Aggarwal, S. and Sabnani, K. (eds.): *Protocol Specification, Testing, and Verification VIII*. Elsevier Science Publishers B.V. (North-Holland), 1988.

[Bar85] Barringer, H.: *A Survey of Verification Techniques for Parallel Programs*. Springer-Verlag Lecture Notes in Computer Science 191, Berlin, 1985.

[BHMY89] Bevier, W.R., Hunt, W.A., Moore, J S. and Young, W.D.: Special issue on system verification. *Journal of Automated Reasoning*, **5**(4), 409–530 (1989).

[BY91] Bevier, W.R. and Young, W.D.: The proof of correctness of a fault-tolerant circuit design. In *Proceedings of the Second International Working Conference on Dependable Computing for Critical Applications*, pp. 107–114. IFIP, February 1991.

[BM79] Boyer, R.S. and Moore, J S.: *A Computational Logic*. Academic Press, New York, 1979.

[BM88] Boyer, R.S. and Moore, J S.: *A Computational Logic Handbook*. Academic Press, New York, 1988.

[BH90] Brock, B.C. and Hunt, W.A.: A formal introduction to a simple hdl. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pp. 285–329. Elsevier Science Publishers B.V. (North-Holland), 1990.

[Cam88] Campbell, J.: *C Programmer's Guide to Serial Communications*. Howard W. Sams and Co., 4300 West 62 Street, Indianapolis, IN 46268, 1988.

[CM88] Chandy, K.M. and Misra, J.: *Parallel Program Design: A Foundation*. Addison Wesley, Massachusetts, 1988.

[CES86] Clarke, E.M., Emerson, E.A. and Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, **8**(2), 244–263 (1986).

[CG87] Clarke, E.M. and Grumberg, O.: Research on automatic verification of finite-state concurrent systems. *Ann. Rev. Comput. Sci.*, **2**, 269–290 (1987).

[DiV81] DiVito, B.L.: A mechanical verification of the alternating bit protocol. Technical Report ICSCA-CMP-21, Institute for Computing Science, The University of Texas at Austin, 1981.

[DiV82] DiVito, B.L.: 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.

[Gol90] Goldschlag, D.M.: Mechanizing unity. In *Programming Concepts and Methods*. North Holland, Amsterdam, 1990.

[Hoa85] Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice Hall International, Englewood Cliffs, NJ, 1985.

[Hun85] Hunt, W.A.: Fm8501: A verified microprocessor. Phd thesis, University of Texas at Austin, December 1985. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

[Int91] Intel Corporation: *Microcommunications*. Intel Literature Sales, P.O. Box 7641, Mt. Prospect, IL 60056-7641, 1991.

[JL91] Jain, P. and Lam, S.S.: Specification real-time protocols for broadcast networks. *IEEE Transactions on Computers*, **40**(4), 404–422 (1991).

[Joy88] Joyce, J.J.: Formal specification and verification of asynchronous processes in higher-order logic. Technical Report No. 136, University of Cambridge Computer Laboratory, June 1988.

[Joy90] Joyce, J.J.: Multi-level verification of microprocessor-based systems. Technical Report No. 195, University of Cambridge Computer Laboratory, May 1990.

[MP84] Manna, Z. and Pnueli, A.: Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, **4**, 257–289 (1984).

[Mil80] Milner, R.: *A Calculus of Communicating Systems*. Springler-Verlag, Berlin, 1980.

[Mis87] Mish, F.C., editor.: *Webster's Ninth New Collegiate Dictionary*. Merriam-Webster, Inc, 1987.

[Moo92a] Moore, J S.: A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. Technical Report NASA CR-4433, NASA, 1992.

[Moo92b] Moore, J S.: Mechanically verified hardware implementing an 8-bit parallel io byzantine agreement processor. Technical Report NASA CR-189588, NASA, 1992.

[PSL80] Pease, M., Shostak, R. and Lamport, L.: Reaching agreement in the presence of faults. *Journal of the ACM*, **27**(2), 228–234 (1980).

[Pet81] Peterson, J.L.: *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.

[Pyg85] Pygott, C.H.: 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.

[Rod88] Roden, M.S.: *Digital Communication Systems Design.* Prentice Hall, 1988.

[RvH89] Rushby, J. and von Henke, F.: Formal verification of the interactive convergence clock synchronization algorithm using ehdm. Technical Report SRI CSL 89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, January 1989.

[Skl88] Sklar, B.: *Digital Communications Fundamentals and Applications.* Prentice Hall, 1988.

[Sun79] Sunshine, C.: Formal techniques for protocol specification and verification. *Computer*, g**12**(9), 20–27 (1979).

[Yoe90] Yoeli, M.: *Formal Verification of Hardware Design.* IEEE Computer Society Press, Los Alamitos, California, 1990.

[You91] Young, W.D.: Verifying the interactive convergence clock synchronization algorithm using the boyer-moore theorem prover. Internal Note 199, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, January 1991.