Accepted at the Haskell Symposium'09

Roll Your Own Test Bed for Embedded Real-Time Protocols: A Haskell Experience

Lee Pike
Galois, Inc.
leepike@galois.com

Geoffrey Brown
Indiana University
geoffrey.brown@acm.org

Alwyn Goodloe
National Institute of Aerospace
Alwyn.Goodloe@nianet.org

Abstract

We present by example a new application domain for functional languages: emulators for embedded real-time protocols. As a case-study, we implement a simple emulator for the Biphase Mark Protocol, a physical-layer network protocol in Haskell. The surprising result is that a pure functional language with no built-in notion of time is extremely well-suited for constructing such emulators. Furthermore, we use Haskell's property-checker QuickCheck to automatically generate real-time parameters for simulation. We also describe a novel use of QuickCheck as a "probability calculator" for reliability analysis.

Categories and Subject Descriptors B.8.1 [*Hardware*]: Performance and Reliability

General Terms Languages, Reliability, Verification

Keywords Physical-layer protocol Testing, Emulation, Functional Programming

1. Introduction

We present by example a new application domain for functional languages: building efficient emulators for real-time systems. Real-time systems are difficult to design and validate due to the complex interleavings possible between executing real-time components. Emulators assist in exploring and validating a design before committing to an implementation. Our goal in this report is to convince the reader by example[1] that

1. one can easily roll-one's-own test bed for embedded real-time systems using standard functional languages, with no built-in notion of real-time;
2. testing infrastructure common to functional languages, such as QuickCheck (Claessen and Hughes 2000), can be exploited to generate real-time parameters for simulation—we generate approximately 100,000 real-time parameters and execution traces per minute on a commodity laptop;
3. and QuickCheck can be used for a novel purpose—to do statistical reliability analysis.

In our report, we assume that the reader is familiar with Haskell syntax. That said, our approach uses basic concepts shared by modern functional languages and does not intrinsically rely on laziness (or strictness) or special monads, for example.

In the remainder of this introduction, we motivate the problem domain and describe related work before going on to describe the emulator framework.

Problem Space: Physical Layer Networking The physical layer resides at the lowest level of the network stack and defines the mechanism for transmitting raw bits over the network. At the physical layer, bits are encoded as voltage signals. A bit stream is transmitted by modulating the electrical signal on an interconnect (e.g., coaxial cable). It is not as simple as translating the 1 to high voltage and 0 to low voltage because the receiver needs to be able to detect when there are consecutive ones or zeros and know when the sender has changed the signal. The inherent complexity at this layer results from (1) the sender and receiver not sharing a hardware clock (so they are asynchronous) and (2) the continuity of the physical world. Thus, the digital abstraction cannot be assumed to hold at this level. Furthermore, we must model the jitter and drift of hardware clocks and the time an electrical signal takes to settle before it stabilizes to a high or low value. If the receiver samples the interconnect at the wrong time, the signal may be misinterpreted by the receiver. The goal is to design a protocol and define timing constraints to ensure the receiver samples the interconnect at the right intervals to reliably decode the bit stream sent by the transmitter.

Many physical protocols exist, but we shall focus on the Biphase Mark Protocol (BMP), which is used to transmit data in digital audio systems and magnetic card readers (e.g., for credit cards). The emulator is modularized: emulating another protocol requires changing just a few small functions (about 30 lines of code).

Background and Related Work Physical layer protocols have been a canonical challenge problem in the formal methods community. Recent work uses decision procedures (more precisely, satisfiability modulo theories) and model-checking to verify their correctness (Brown and Pike 2006); these results compare favorably to previous efforts using mechanical theorem-proving, which required thousands of manual proof steps (Moore 1994; Vaandrager and de Groot 2004). Indeed, the emulator described here is essentially refined from its high-level specification in a model checker (Brown and Pike 2006). Given the success of these formal verification techniques—which *prove* correctness—what interest is there in simulation?

There are at least a few responses. To begin with, it is not always the case that the constraints can be expressed in a decidable theory. In particular, timing constraints that contain non-linear inequalities

[1] The source code associated with this paper is presented in the Appendix and is also available for download at `http://www.cs.indiana.edu/~lepike/pub_pages/qc-biphase.html`. The code is released under a BSD3 license. The emulator is about 175 lines of code, and the QuickCheck infrastructure is about 100 lines.

cannot be decided (in this case, it so happens that our expression of the BMP constraints are linear). Furthermore, decision procedures and model-checkers are complex and may contain bugs, or the model itself may contain bugs. Both cases may lead to vacuous proofs, but because the "execution" of a model-checker's model is symbolic, it can be difficult to sanity-check the correctness of the model or tool. An emulator, however, is executed on concrete data. Another motivation is that even if there are no bugs in a formal model, a proof of correctness is only as good as the connection between the model used in the proof and its fidelity to the implementation. The components of a Haskell emulator can be, in principle, refined into digital hardware (Sheeran 2005), and the QuickCheck-generated data can be used not only to drive the emulator, but as test-vectors for the implemented hardware. Finally, as we discuss in Section 5, QuickCheck can be used as a "probability calculator" for reliability analysis of digital systems, something that cannot be done easily with current formal verification tools.

The work described here is part of a larger framework being developed by the two authors Pike and Goodloe for the purpose of building emulators for real-time safety-critical distributed systems under a NASA contract. On top of the emulator described here, we have built infrastructure to simulate a serial broadcast bus with multiple receivers and cyclic redundancy checks over the data by the receivers. Functional languages make constructing the additional emulator machinery easy; for example, a serial bus emulator is constructed by doing little more than mapping the emulator described here over a list of receivers.

2. Biphase Mark Protocol (BMP)

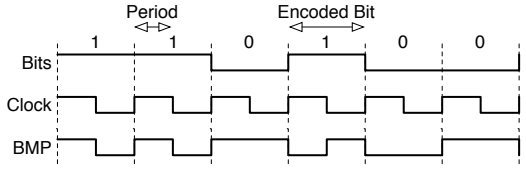

Figure 1. BMP Encoding of a Bit Stream

We begin by describing the protocol. The simple portion of the protocol is the encoding of a bit stream by the transmitter. Consider Figure 1, where the top stream is the bit stream to be transmitted and the middle stream is the transmitter's clock. In BMP, every encoded data bit is guaranteed to begin with a transition marking a *clock event*; that is, the transmitter begins an encoded bit by modulating the signal on the interconnect. The value of the encoded bit is determined by the presence (to encode a 1) or absence (to encode a 0) of a transition in the middle of the encoded bit. Thus, a 0 is encoded as either two sequential low or high signals (e.g., 00 or 11), while a 1 is encoded as either a transition from high to low or low to high (e.g., 01 or 10).

The central design issue for the receiver is to extract a clock signal from the combined signal reliably. The receiver has two modes, a *scanning* mode in which it attempts to detect a clock event marking the first half of an encoded bit, and a *sampling* mode in which it assumes that sufficient synchrony has been established to simply sample the signal at some point while the second half of the bit is being transmitted.

In each of these modes, real-time constraints must be met to ensure correct operation. To see why, consider Figure 2 which represents a hypothetical plot over time of the strength of a signal sent by a transmitter. The *period* is the nominal interval between clock signal transitions, as shown in Figure 1. For some portion of the

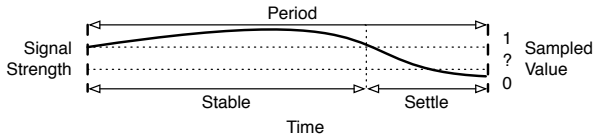

Figure 2. Signal Strength Over Time

period, the signal is stable. During the stable interval, the signal is guaranteed to be sufficiently high or low (in the figure, it is high) so that if the receiver samples the signal then, it is guaranteed to be sampled correctly. During the remainder of the period, however, the signal is settling, so the receiver nondeterministically interprets the signal as high, low, or indeterminate.

The real-time constraints on when the receiver scans and samples, described in the following section, are the key to the protocol correctness.

3. Real-Time Parameters and Constraints

We approximate dense real-time using double-precision floating point numbers in Haskell:

```
type Time = Double
```

Real-time parameters associated with transmitter and receiver are captured in a data type. Simulation runs are executed over instances of this data type. (We affix a 't' or 'r' to the parameter names to remind ourselves whether they're associated with the transmitter, `tx`, or receiver, `rx`.)

```
data Params = Params
    { tPeriod  :: Time -- ^ Tx's nominal clock period.
    , tSettle  :: Time -- ^ Maximum settling time.
    , rScanMin :: Time -- ^ Rx's min scan duration.
    , rScanMax :: Time -- ^ Rx's max scan duration.
    , rSampMin :: Time -- ^ Rx's min sampling duration.
    , rSampMax :: Time -- ^ Rx's max sampling duration.
    } deriving (Show, Eq)
```

The field `tPeriod` contains the nominal period of the transmitter. The field `tSettle` contains the maximum settling duration for the signal—we use the *maximum* possible settling interval so that the model is as pessimistic as possible, since the value of the signal is indeterminate while settling. (We do not need to keep track of `tStable` since we can compute it by `tPeriod - tSettle`.) We then have fields containing the minimum and maximum real-time values that bound the intervals of time that pass between successive scanning or sampling by the receiver. The difference between the minimum and maximum values captures the error introduced by clock drift and jitter. Indeed, these bounds are used to capture the cumulative error in *both* the transmitter's and receiver's clock. By ascribing the cumulative error to the receiver in the model, we can assume the transmitter's clock is error-free and always updates at its nominal period—otherwise, we would have fields recording minimum and maximum `tPeriod` intervals—so it is a modeling convenience.

We can now define a relation containing a conjunction of constraints over the parameters that (we hope!) ensure correct operation. These timing constraints are at the heart of what makes demonstrating the correctness of physical layer protocols difficult.

```
1  correctParams :: Params → Bool
2  correctParams p =
3       0 < tPeriod p
4    && 0 ≤ tSettle p
5    && tSettle p < tPeriod p
6    && 0 < rScanMin p
7    && rScanMin p ≤ rScanMax p
8    && rScanMax p < tStable
9    && tPeriod p + tSettle p < rSampMin p
10   && rSampMin p ≤ rSampMax p
11   && rSampMax p < tPeriod p + tStable - rScanMax p
12     where tStable = tPeriod p - tSettle p
```

Some of the constraints are simply "sanity constraints" to ensure time is positive (e.g., the constraints on lines 3, 4, and 6) or that a minimum bound is no greater than a corresponding maximum bound (e.g., the constraints on lines 7 and 10). The other constraints are more interesting and derive from a designer's domain knowledge regarding the protocol. For example, the constraint on line 9 ensures that even if `rx` detects the first half of an encoded bit too early (i.e., just after it starts modulating at the beginning of the settling interval), it waits until the end of the settling interval plus the entire period (containing the stable interval of the first half of the bit and the settling interval of the second half of the bit) before sampling. This ensures `rx` does not sample before the stable interval of the period containing the second half of the bit.

These constraints are complex and we want to simulate the protocol's execution to ensure they are correct and if they are, that our implementation satisfies them.

4. The Emulator

So far, we have described the protocol and the real-time constraints we posit it must satisfy. To simulate it, we need an executable model. We begin by describing a model of real-time for the emulator then the emulator itself.

4.1 Model of Time

Our model of time borrows from the discrete-event simulation model (Dutertre and Sorea 2004; Schriber and Brunner 1999). In this model, each independent real-time component, C, in a system possesses a *timeout* variable that ranges over `Time`. That timeout variable denotes the point in time at which C will make a state transition. The value of C's timeout variable is always in the future or the present; when it is at the present, C exercises a state transition, and its timeout variable is updated (possibly nondeterministically) to some point strictly in the future.

In our case, the transmitter and receiver each possess a timeout variable, which we denote as `tclk` and `rclk`, respectively. Intuitively, these values "leap frog" each other. The least-valued timeout is considered to be at the present, and so that component executes. Of course, one timeout might be significantly less than the other and will make successive transitions before the other component possesses the least-valued timeout.

The primary advantage of this model of time is that it is simple: we do not need a special semantics to model real-time execution.

4.2 Emulator Architecture

In Figure 3, we show an abstract representation of the system as it is modeled. We describe the components below.

The Transmitter The transmitter is comprised of three Haskell functions (and some small helper functions): an environment `tenv`, encoder `tenc`, and the transmitter's clock, `tclock`. Of these, only the encoder is protocol-specific; the remainder are generic infrastructure.

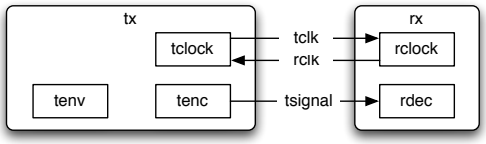

Figure 3. Emulator Architecture

The environment `tenv` simply returns a new random bit to send. Regarding the timeout function `tclock`, recall from Section 3 that in our model, we attribute errors to the receiver. Thus, transmitter's timeout is updated deterministically: each application of `tclock` update's `tx`'s timeout by exactly `tPeriod p`. This leaves only the transmitter's encoder `tenc`. This function is the protocol-specific portion of the transmitter's definition. The function has three possible branches. If the transmitter is not in the middle of sending an encoded bit, it may nondeterministically (using the `System.Random` library) idle the signal (i.e., not modulate the signal), or it may send the first half of an encoded bit. Otherwise, it encodes the second half of a bit.

The Receiver Architecturally, the receiver is simpler than the transmitter since it only contains a clock and a decoder. However, both of their definitions are more complex: `rx`'s clock is more complex because we capture the effects of drift, jitter, and so forth here, so the timeout updates nondeterministically (again using the `System.Random` library); `rx`'s decoder is more complex because here we model whether `rx` captures the signal depending on the relationship between `tx`'s and `rx`'s timeouts.

The receiver's timeout function updates the timeout nondeterministically depending on which of two modes `rx` is in. If `rx` is expecting the first half of an encoded bit (so in its scanning mode), it updates the timeout `rclk` to some random value within the inclusive range `[rclk + rScanMin p, rclk + rScanMax p]`, where `p` is an instance of `Params` defined in Section 3. If `rx` is in the sampling mode, it similarly updates its timeout to some random value within `[rclk + rSampMin p, rclk + rSampMax p]`.

As mentioned, the decoder `rdec` is where we model the effects of incorrectly sampling the signal. The decoder follows the BMP protocol to decode an incoming signal if `stable` is true, and fails to detect the signal properly otherwise. The function `stable` takes `rx`'s and `tx`'s state (implemented as data types) and returns a boolean:

```
stable :: Params → Rx → Tx → Bool
stable p rx tx =
     not (changing tx)
  || tclk tx - rclk rx < tPeriod p - tSettle p
```

Recall that `tclk` and `rclk` are the timeouts. The value of `changing tx` is a boolean that is part of `tx`'s state—it is true if `tx` is modulating the signal in the next period. Thus, the function `stable` is true if either the signal is not going to modulate (so that even if it is sampled during the settling interval, it is sampled correctly), or the receiver's timeout falls within the stable interval—recall Figure 2. If `stable` is false, we return the opposite value of the signal being sent by the transmitter. This ensures our emulator is overly-pessimistic and captures potentially metastable events even if they may not result in a faulty signal capture in reality.

Wiring The Transmitter and Receiver Together The function `transition` causes either `tx` or `rx` to execute a state-update. The function takes a set of real-time parameters, the receiver's and transmitter's states, and return new states (within the `IO` monad).

```
transition :: Params → Rx → Tx → IO (Rx, Tx)
transition p rx tx
  | tclk tx ≤ rclk rx
    = do tx' ← txUpdate p tx
                return (rx {synch = False}, tx')
  | otherwise
    = do rx' ← rxUpdate p rx tx
                return (rx', tx)
```

The `txUpdate` function updates `tx`'s state by applying the functions `tenv`, `tenc`, and `tclock`. Likewise for `rxUpdate`, except `rxUpdate` takes `tx`'s state too, as based on the relationship between `tx`'s timeout and its own, it may sample the signal correctly or not. Whether `tx` or `rx` is updated depends on which timeout is least—if they are equal, we arbitrarily choose to update `tx`'s state.

Executing this function takes one "step" of the discrete-event emulator. We initialize the state of the transmitter and receiver, and then iteratively call the `transition` function for some user-specified number of rounds.

5. QuickCheck: Automatically Generating Timing Parameters

QuickCheck is a popular tool for automatically testing programs. Because our emulator itself generates random values (e.g., timeout updates for `rx`), the emulator executes within the `IO` monad; therefore, we use a monadic extension of QuickCheck (Claessen and Hughes 2002).

Test-Case Generation Our first task is to generate parameters that satisfy the `correctParams` function defined in Section 2. The naïve approach is to generate random instances of the `Params` data type and throw away those instances that do not satisfy `correctParams`. Unfortunately, this approach generates almost no satisfying instances because so few random parameters satisfy the constraints.

Therefore, we define a custom generator. However, we have the following problem: the set of inequalities in `correctParams` are circular and not definitional. The conjuncts of `Params` cannot be placed in a linear order such that each constraint introduces no more than one new parameter. Thus, we cannot sequentially generate parameters that satisfy them.

Our solution is to define a generator that over-approximates the inequalities in `correctParams`. For example, we can replace any occurrence of the parameter `tSettle p` on the right-hand side of $\leq$ with the parameter `tPeriod p`, since the latter is guaranteed to be larger than the former. By over-approximating, we can rewrite the inequalities so that each constraint introduces just one new parameter. This over-approximation is "close enough" so that a large number of generated instances satisfy `correctParams`—we can then prune out the few instances that do not satisfy `correctParams`.

Validation The following is the fundamental correctness property we wish to validate: whenever the receiver has captured (what it believes to be) the second half of an encoded bit, the bit it decodes is the one that `tx` encoded. (Again, `Rx` and `Tx` are the data types containing the receiver's and transmitter's respective state.)

```
bitsEq :: Rx → Tx → Bool
bitsEq rx tx = tbit tx == rbit rx
```

In the property, `tbit tx` is the bit that `tx` is encoding, and `rbit rx` is the bit `rx` has decoded.

QuickChecking this property over millions of simulation runs suggests (but of course does not prove) that our parameters are indeed correct. And it is fast. On a commodity laptop (MacBook Pro, 2.5 GHz Intel Core 2 Duo with 4 GB of memory), our emulator automatically generates approximately 100,000 simulations of the protocol in a minute.[2]

As with emulators in other programming languages, the efficacy of our test-bed for discovering timing errors is contingent upon the number of and duration of test runs, the coverage achieved by the generated test data, and the significance of the timing violation.

QuickCheck as a Probability Calculator In standard practice, QuickCheck is used to validate a property and to return a counterexample otherwise. This usage model makes sense when verifying that programs operate correctly over discrete data such as lists, trees, and integers. In real-time systems, however, we identify a novel usage of QuickCheck as a probability calculator.

For (a slightly contrived) example, suppose that for some legacy hardware configuration, we know that the settling interval is no more than 5% of the period, and the receiver's bounds on scanning and sampling ensure it consistently captures the data. Later, suppose the receiver is to be used in a new configuration in which the settling interval may be up to 15% of the period. The receiver's bounds on scanning and sampling cannot be changed, since they are determined by its legacy clock. Now we ask what percentage of bits will the receiver incorrectly decode?

To answer this question, we generate a fixed number of tests and determine what percentage of them fail. To facilitate this use of QuickCheck, we slightly extend its API.[3] For the example described, generating 100,000 tests results in a failure rate (i.e., the property `bitsEq` above fails) of approximately 0.2%. Depending on the performance of error-checking codes and other constraints, this bit-error rate may be satisfactory.

Another use of QuickCheck as a "probability calculator" is to compute the probability of cyclic redundancy checks capturing bit-transmission errors under different fault scenarios (Driscoll et al. 2003; Paulitsch et al. 2005). In general, this appears to be a powerful application of QuickCheck for testing stochastic systems.

Using QuickCheck as a probability calculator depends on QuickCheck generating a sufficiently large number of appropriately-distributed tests. We have not verified the extent to which this hypothesis holds in various domains.

6. Conclusion

In this report, we demonstrate via example that functional languages—particularly Haskell—and their associated tools (i.e., QuickCheck) are unexpectedly well-suited to build real-time emulators. We have applied QuickCheck in two new ways—to generate real-time parameters and as a probability calculator for reliability analysis. We hope this report motivates others to explore the use of functional programming for building emulation test-beds for real-time systems.

Acknowledgments

This work is supported by NASA Contract NNL08AD13T from the Aviation Safety Program Office. We thank for the following individuals for their advice and guidance on this work: Ben Di

[2] These performance results use a single core and suppress output to standard out. While there are no special performance optimizations made to the code, we use the `System.Random.Mersenne` Haskell library for fast random-number generation.

[3] A corresponding patch is available at `http://www.cs.indiana.edu/~lepike/pub_pages/qc-biphase.html`.

Vito of the NASA Langley Research Center; Levent Erkok, Dylan McNamee, Iavor Diatchki, and Don Stewart, and John Launchbury of Galois, Inc.; Rebekah Leslie of Portland State University; and Andy Gill of the University of Kansas; and Bastiaan Heeren of the Open Universiteit Nederland.

References

Geoffrey M. Brown and Lee Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2006. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/bmp.html`.

Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM Press, 2000.

Koen Claessen and John Hughes. Testing monadic code with QuickCheck. In *In Proc. ACM SIGPLAN workshop on Haskell*, pages 65–77, 2002.

Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *Computer Safety, Reliability, and Security*, LNCS, pages 235–248. SAFECOMP, Springer-Verlag, September 2003.

Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 3253 of *LNCS*. Springer-Verlag, 2004.

J Strother Moore. A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. *Formal Aspects of Computing*, 6(1):60–91, 1994. URL `citeseer.ist.psu.edu/moore92formal.html`.

Michael Paulitsch, Jennifer Morris, Brendan Hall, Kevin Driscoll, Elizabeth Latronico, and Philip Koopman. Coverage and the use of cyclic redundancy codes in ultra-dependable systems. In *International Conference on Dependable Systems and Networks (DSN 2005)*, pages 346–355, 2005.

Thomas J. Schriber and Daniel T. Brunner. Inside discrete-event simulation software: how it works and why it matters. In *Winter Simulation Conference*, pages 72–80, 1999.

M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7): 1135–1158, 2005.

F. W. Vaandrager and A. L. de Groot. Analysis of a Biphase Mark Protocol with Uppaal and PVS. Technical Report NIII-R0455, Nijmegen Institute for Computing and Information Science, 2004.

A. Biphase.hs

```
module Biphase where

-- A faster random-number generator
import System.Random.Mersenne

---------- DATATYPES ---------------------------------------
type Time = Double

-- | Realtime input parameters.
data Params = Params
    { tPeriod  :: Time -- ^ Tx's clock period.
    , tSettle  :: Time -- ^ Nominal signal settling time.
    , rScanMin :: Time -- ^ Rx's min scan duration.
    , rScanMax :: Time -- ^ Rx's max scan duration.
    , rSampMin :: Time -- ^ Rx's min sampling duration.
    , rSampMax :: Time -- ^ Rx's max sampling duration.
    } deriving (Show, Eq)


data TState = SendFirst -- ^ Sending the 1st datum;
            | SendSecond -- ^ Sending the 2nd.
              deriving (Show, Eq)

data Tx = Tx
    { tstate   :: TState -- ^ Tx's state.
    , tsignal  :: Bool -- ^ Signal being sent.
    , tbit     :: Bool -- ^ Encoded bit to be sent.
    , changing :: Bool -- ^ T: modulating the signal; F o/w.
    , tclk     :: Time -- ^ Tx's timeout.
    } deriving (Show, Eq)

data RState = RcvFirst -- ^ Expecting the 1st datum;
            | RcvSecond -- ^ Expecting the 2nd.
              deriving (Show, Eq)

data Rx = Rx
    { rstate  :: RState -- ^ Rx's state.
    , rsignal :: Bool -- ^ Current datum being received.
    , rbit    :: Bool -- ^ Decoded bit.
    , rclk    :: Time -- ^ Rx's timeout.
    , synch   :: Bool -- ^ Rx just transitioned from
                      -- RcvSecond to RcvFirst
                      -- (capturing a bit).
    } deriving (Show, Eq)

------------------------------------------------------------

-- Helper for Mersenne randoms
randomRng :: (Time, Time) → IO Time
randomRng (low, high) = do r ← randomIO
                           return $ low + (r * (high - low))

---------- INITIAL STATE/CLOCKS ----------------------------
initTx :: Params → IO Tx
initTx p = do t ← randomRng (0, tPeriod p - tSettle p)
              bit ← randomIO
              return Tx { tstate = SendFirst
                        , tsignal = True
                        , tbit = bit
                        , changing = False
                        , tclk = t}

initRclock :: Params → IO Time
initRclock p = do r ← randomRng (0, rScanMax p)
                  -- we want a random in [a, a)
                  if r == rScanMax p
                    then initRclock p
                    else return r

initRx :: Params → IO Rx
initRx p = do r ← initRclock p
              bit ← randomIO
              return Rx { rstate = RcvFirst
                        , rsignal = True
                        , rbit = bit
                        , rclk = r
                        , synch = False}
------------------------------------------------------------

---------- Tx UPDATE ---------------------------------------
-- |
tenv :: Tx → IO Tx
tenv tx = case tstate tx of
            SendFirst → do ran ← randomIO
                           return tx {tbit = ran}
            SendSecond → return tx
```

```
-- | The transmitter's encoder.  Protocol-specific.
tenc :: Tx → IO Tx
tenc tx =
    case tstate tx of
      SendFirst →
          do idle ← randomIO
             if idle -- Idling
                then return tx {changing = False}
                -- 1st half of a new bit.
                else return
                          tx { tsignal = ttoggle
                               , tstate = SendSecond
                               , changing = True}

      SendSecond → return tx { tsignal = toggle
                               , tstate = SendFirst
                               , changing = changed toggle}
    where toggle = if tbit tx
                   then ttoggle else tsignal tx
          ttoggle = not $ tsignal tx
          changed cur = cur /= tsignal tx

tclock :: Params → Tx → Tx
tclock p tx = tx {tclk = tPeriod p + tclk tx}

txUpdate :: Params → Tx → IO Tx
txUpdate p tx = do
  tx' ← tenv tx
  tx'' ← tenc tx'
  return $ tclock p tx''
------------------------------------------------------------

---------- Rx UPDATE -----------------------------------
-- | Correct update of rclk---helper
rclock :: Params → Rx → IO Time
rclock p rx =
    let r = rclk rx
    in case rstate rx of
         RcvFirst →
             randomRng (r + rScanMin p, r + rScanMax p)
         RcvSecond →
             randomRng (r + rSampMin p, r + rSampMax p)

stable :: Params → Rx → Tx → Bool
stable p rx tx =
     not (changing tx)
  || tclk tx - rclk rx < tPeriod p - tSettle p

-- | The receiver's decoder.  Protocol-specific.
rdec :: Params → Rx → Tx → Rx
rdec p rx tx =
      -- Are we in a "stable" part of the signal?
  let badSignal = not $ tsignal tx
      v         = if stable p rx tx
                  then tsignal tx else badSignal
  in case rstate rx of
       RcvSecond →  rx { rsignal = v
                       , rbit = rsignal rx /= v
                       , rstate = RcvFirst}
       RcvFirst  → rx { rsignal = v
                      , rstate = signal}
           where signal = if v == rsignal rx
                          then RcvFirst
                          else RcvSecond

rxUpdate :: Params → Rx → Tx → IO Rx
rxUpdate p rx tx = do
  let rx'     = rdec p rx tx
      rchange = case (rstate rx, rstate rx') of
                  (RcvSecond, RcvFirst) → True
                  _                     → False
  r ← rclock p rx'
  return rx' { rclk = r
             , synch = rchange}
------------------------------------------------------------

-- | Full state transition.
transition :: Params → (Rx, Tx) → IO (Rx, Tx)
transition p (rx, tx)
  | tclk tx ≤ rclk rx = do
      tx' ← txUpdate p tx
      return (rx {synch = False}, tx')
  | otherwise = do
      rx' ← rxUpdate p rx tx
      return (rx', tx)

putLnState :: Integer → (Rx, Tx) → IO ()
putLnState i (rx, tx) = do
  putStrLn $ "States: " ++ (show $ tstate tx) ++ " "
               ++ (show $ rstate rx)
  putStrLn $ "Clocks: "
               ++ (show $ tclk tx) ++ " "
               ++ (show $ rclk rx)
  putStrLn $ "Bits: "
               ++ (show $ tbit tx) ++ " "
               ++ (show $ rbit rx)
               ++ " Signal: " ++ (show $ tsignal tx)
               ++ " " ++ (show $ rsignal rx)
  putStrLn $ "i: " ++ (show i) ++ " Synch: "
               ++ (show $ synch rx) ++ "λn"

-- | Defines a "good" stop state: tx has sent the 2nd
-- signal bit and rx has sampled it.
stopState :: Rx → Bool
stopState rx = synch rx

execToStopState :: Bool → Params → Integer → (Rx, Tx) →
IO (Rx, Tx)
execToStopState output p i s = do
    if output then putLnState i s else return ()
    if stopState (fst s)
      then return s
      else execToStopState output p i =≪ transition p s

-- | Exectuion of the protocol.
exec :: Bool → Params → Integer → (Rx, Tx) → IO (Rx, Tx)
exec output p i s = do
  s' ← execToStopState output p i s
  if i < 1 then return s'
           else exec output p (i-1) s'

-- | Begin a finite trace of length i from the initial
-- state. Either send one determined signal bit or a
-- series of nondeterministic signals.
startExec :: Bool → Params → Integer → IO (Rx, Tx)
startExec output p i = exec output p i =≪ initState p

-- | The initial state.
initState :: Params → IO (Rx, Tx)
initState p = do
  rx ← initRx p
  tx ← initTx p
  return (rx, tx)
```

B. BiphaseQC.hs

```
module Main

where

import Biphase
import Test.QuickCheck
import Test.QuickCheck.Monadic
import Test.QuickCheck.Gen
```

```
-- | Number of rounds to execute
iter :: Integer
iter = 1

-- | Property should always hold for good parameters.
prop_correct :: Bool → Property
prop_correct output =
    assertFinal output forallValidParams bitsEq

-- | Testing should fail on this property for some
-- percentage of tests.
prop_incorrect :: Bool → Property
prop_incorrect output =
    assertFinal output forallInvalidParams bitsEq

-- | Did the receiver get the bits sent by the sender upon
-- synchronizing?
bitsEq :: (Rx, Tx) → Bool
bitsEq (rx, tx) = (tbit tx) == (rbit rx)

-- | Note: monadicIO (from QuickCheck) uses unsafeperformIO.
assertFinal :: Bool → ParamGen → ((Rx, Tx) → Bool) →
Property
assertFinal output genParams pred =
    monadicIO $ genParams $ λp →
        assert ∘ pred =≪ run (startExec output p iter)

---------- SIMPLE MAIN FUNCTION (modify as needed) -------
main = do
  putStrLn ""
  putStrLn $ "Enter the number of bits to encode"
       ++ " (an integer between 1 and 100 million): "
  s ← getLine
  putStrLn $ "Show output for each test? (True or False)"
  output ← getLine
  let i = read s
    in if i < 1 || i > 10^8
         then main
         else quickCheckQuotientWith stdArgs {maxSuccess =
i} (prop_correct $ read output)

-------------------------------------------------------

type ParamGen =
    (Params → PropertyM IO ()) → PropertyM IO ()

-- | Generating correct params is "too hard" to do
-- procedurally, so we get close and then use a
-- predicate to make sure we're only testing correct ones.
forallValidParams :: ParamGen
forallValidParams =
    forAllM (genParams `suchThat` correctParams)

-- | Generate *almost* correct realtime parameters --- it's an
-- overapproximation.  We need to test them to ensure
-- correctness.
genParams :: Gen Params
genParams = do
  -- arbitrary-sized clock period
  tperiod ← choose (0, 100)
  -- The remaining generated values are over-approximations.
  tsettle ← choose (0, tperiod)
  rscanmin ← choose (0, tperiod - tsettle)
  rscanmax ← choose (rscanmin, tperiod)
  rsampmin ← choose ( tperiod + tsettle
                    , 2 * tperiod - tsettle - rscanmax)
  rsampmax ← choose ( rsampmin
                    , 2 * tperiod - tsettle - rscanmax)
  return $ Params tperiod tsettle rscanmin
                  rscanmax rsampmin rsampmax

-- | Constraints are satisfied.  Reproduced for genParams.
correctParams :: Params → Bool
correctParams p =
     0 < tPeriod p -- tPeriod
  && 0 ≤ tSettle p -- tSettle
  && tSettle p < tPeriod p -- tSettle
  && 0 < rScanMin p -- rScanMin
  && rScanMin p ≤ rScanMax p -- rScanMax
  && rScanMax p < tStable  -- rScanMax
  && tPeriod p + tSettle p < rSampMin p -- rSampMin
  && rSampMin p ≤ rSampMax p -- rSampMax
  -- rSampMax
  && rSampMax p < tPeriod p + tStable - rScanMax p
   where tStable = tPeriod p - tSettle p

--- GENERATING FAILING TESTS ---

forallInvalidParams :: ParamGen
forallInvalidParams =
    forAllM (badGenParams `suchThat` incorrectParams)

-- Example in hte paper.
badGenParams :: Gen Params
badGenParams =
    let tperiod = 100
        tsettleNewMax = 15
        tsettle = 5
    in do
  -- arbitrary-sized clock period
  --  tperiod ← choose (0, 100)
  -- The remaining generated values are over-approximations.
  tsettle' ← choose (0, tsettleNewMax)
  rscanmin ← choose (0, tperiod - tsettle)
  rscanmax ← choose (rscanmin, tperiod)
  rsampmin ← choose ( tperiod + tsettle
                    , 2 * tperiod - rscanmax - tsettle
                    )
  rsampmax ← choose ( rsampmin
                    , 2 * tperiod - rscanmax - tsettle
                    )
  return $ Params tperiod tsettle' rscanmin
                  rscanmax rsampmin rsampmax


-- Constraints are satisfied.  Reproduced for genParams.
incorrectParams :: Params → Bool
incorrectParams p =
    let tSettle' = 5
    in    0 < tPeriod p -- tPeriod
       && 0 ≤ tSettle' --p -- tSettle
       && tSettle p < tPeriod p -- tSettle p
       && 0 < rScanMin p -- rScanMin
       && rScanMin p ≤ rScanMax p -- rScanMax
       && rScanMax p < tPeriod p - tSettle' -- p -- rScanMax
       &&   tPeriod p + tSettle'  -- p
          < rSampMin p -- rSampMin
       && rSampMin p ≤ rSampMax p -- rSampMax
       && rSampMax p <  -- p -- rSampMax
              2 * tPeriod p - rScanMax p - tSettle'
```