To appear in the *Participants's Proceedings of DCC: the Seventh International Workshop on Designing Correct Circuits* (Satellite Workshop of ETAPS), 2006.

"Easy" Parameterized Verification of Cross Clock Domain Protocols

Geoffrey M. Brown[1] and Lee Pike[2*]

[1] Indiana University, Bloomington
geobrown@cs.indiana.edu
[2] Galois Connections, Inc.
leepike@galois.com

Abstract. This paper demonstrates how an off-the-shelf model checker that utilizes a Satisfiability Modulo Theories decision procedure and k-induction can be used for verification applications that have traditionally required special purpose hybrid model checkers and/or theorem provers. We present fully parameterized proofs of two types of protocols designed to cross synchronous boundaries: a simple data synchronization circuit and a serial communication protocol used in UARTs (8N1). The proofs were developed using the SAL model checker and its ICS decision procedures.

1 Introduction

This paper uses the bounded model checker and ICS decision procedures of SAL to develop fully parameterized *proofs* of two types of protocols designed to cross synchronous boundaries: a simple data synchronization circuit and a serial communication protocol, 8N1, used in UARTs.[3] Protocols such as these present challenging formal verification problems because their correctness requires reasoning about interacting time events. The proofs discussed in this paper are parameterized by expressing temporal constraints as a system of linear equations. The proofs are "easy" in that they require few proof steps. For example, we have previously presented a proof of the biphase mark protocol [17], which is structurally similar to, though simpler than, 8N1. Our biphase mark proof required 5 invariants, whereas a published proof using PVS required 37; our proof required 5 proof directives (the proof of each invariant is automated), whereas the PVS proof initially required more than 4000 proof directives [1]. Our proofs are quick to check – a few minutes computing time, while one published proof of biphase mark required five hours. Furthermore, our proofs identified a potential bug: in verifying the 8N1 decoder, we found a significant error in a published application note that incorrectly defines the relationship between various real time parameters which, if followed, would lead to unreliable operation [2].

[*] The majority of this work was completed while this author was a member of the Formal Methods Group at the NASA Langley Research Center in Hampton, Virginia.

[3] The SAL specifications and proofs are available at `http://www.cs.indiana.edu/~lepike/pub_pages/dcc.html`.

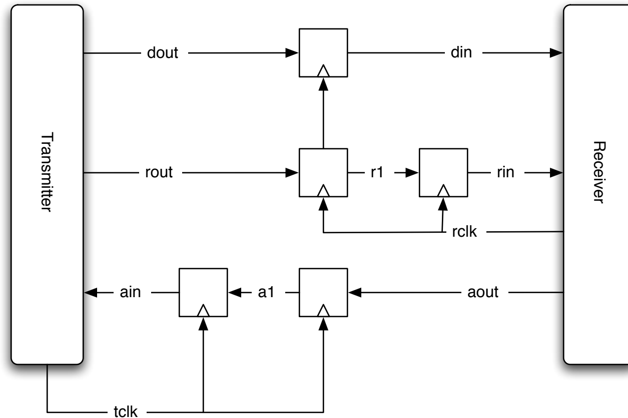

Fig. 1. Synchronizer Circuit

The synchronizer circuit considered in this paper, illustrated in Figure 1, is constructed entirely of D-type flip-flops. The circuit, which is commonly used, allows a transmitter in one clock domain to reliably transmit data to a receiver in another clock domain irrespective of the relative frequencies of the clocks controlling the digital circuitry. This circuit allows the transmitter to send a bit (or in general a word) of data to the receiver through an exchange of "request" (`rout`, `rin`) and "acknowledgment" signals (`aout`, `ain`). A temporal illustration of the exchange between transmitter and receiver is presented in Figure 2. Each event initiated by the transmitter must propagate to the receiver and a response must be returned before the transmitter can initiate a new transfer. The protocol followed by the transmitter and receiver is a simple token passing protocol where the transmitter has the token and hence is allowed to modify its outputs only when `ain = rout`, and the receiver has the token and is allowed to read its input data `din` when `rin != aout`. For example, the transmitter sends data when `rout = ain` by setting `dout` to the value that it wishes to send and by changing the state of `rout`. Informally, the circuit satisfies a simple invariant:

$$rin \neq aout \Rightarrow din = dout \tag{1}$$

Although the protocol is trivial, there is a fundamental issue that greatly complicates the behavior of the circuit – metastability. The fact that the two clocks `rclk` and `tclk` are not synchronized and may run at arbitrary relative rates means that we cannot treat the flip-flops in the circuit as simple delay elements. In particular, the correct behavior of a flip-flop depends upon assumptions about when its input may change relative to its clock. Changes occurring too soon before a clock event are said to violate the "setup time" requirement of the flip-flop while changes occurring too soon after a clock event are said to

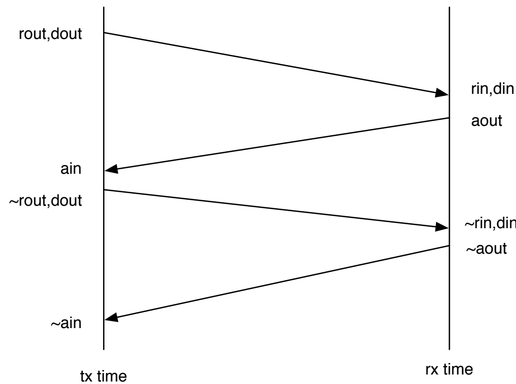

Fig. 2. Synchronizer Circuit Timeline

violate the "hold time" requirement. Either violation may cause the flip-flop to enter a metastable state in which its output is neither "one" nor "zero" and which may persist indefinitely. In practice, probabilistic bounds may be calculated which define how long a metastable state is likely to persist. The illustrated circuit assumes that the time between two events on a single clock is long enough to ensure that the metastability resolution time (plus setup time) is shorter that the clock period with sufficiently high probability. While there have been other proofs of this circuit, they did not model the effects of metastability [3, 4]. An alternative approach has been proposed and is evidently used in a commercial tool to reproduce synchronization bugs by introducing random one-clock jitter in cross domain signals [5, 6]. A fundamental difference between our work and those cited is that we explicitly model timing effects and rely upon clearly stated timing assumptions to verify the circuit.

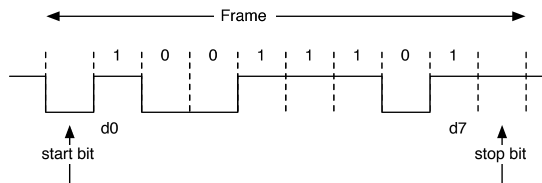

Fig. 3. 8N1 Data Transmission

Metastability also is an issue in the behavior of the 8N1 implementation in which a receiver must sample a changing signal in order to determine the boundaries between valid data. To motivate the design of the 8N1 protocol,

consider Figure 3 which illustrates the encoding scheme utilized by this protocol. In a synchronous circuit, the data and clock are typically transmitted as separate signals; however, this is not feasible in most communication systems (e.g., serial lines, Ethernet, SONET, Infrared) in which a single signal is transmitted. A general solution to this problem is to merge the clock and data information using a coding scheme. The clock is then recreated by synchronizing a local reference clock to the transitions in the received data. In 8N1 a transition is guaranteed to occur only at the beginning of each *frame*, a sequence of bits that includes a start bit, eight data bits, and a stop bit. Data bits are encoded by the identity function – a 1 is a 1 and a 0 is a 0. Consequently, the clock can only be recovered once in each frame in which the eight data bits are transmitted.

Thus, the central design issue for a *data decoder* is reliably extracting a clock signal from the combined signal. Once the location of the clock events is known, extracting the data is relatively simple. Although the clock events have a known relationship to signal transitions, detecting these transitions precisely is usually impossible because of distortion in the signal around the transitions due to the transmission medium, clock jitter, and other effects. A fundamental assumption is that the transmitter and receiver of the data do not share a common time base and hence the estimation of clock events is affected by differences in the reference clocks used. Constant delay is largely irrelevant; however, transition time and variable delay (e.g., jitter) are not. Furthermore, differences in receiver and transmitter clock phase and frequency are significant. Any correctness proof of an 8N1 decoder must be valid over a range of parameters defining limits on jitter, transition time, frequency, and clock phase. Finally, any errors in detection can lead to metastable behavior as with the synchronization circuit.

The temporal proofs presented in this paper may be reproducible using specialized real-time verification tools such as Hytech, TReX and Parameterized Uppaal (we leave it as an open challenge to these respective communities to reproduce these models and proofs in the those tools) [7–9]. However, a key difference is that SAL is a general purpose model checking tool and the real time verification we performed utilized the standard decision procedures. Furthermore, the proofs are not restricted to finite data representations – in the case of the data synchronization circuit our proofs are valid for arbitrary integer data.

The remainder of the paper is organized as follows. In Section 2, we overview the language and proof technology of SAL. The modeling and verification of the synchronizer circuit is presented in Section 3. The model of the 8N1 protocol is presented in Section 4, and its verification is described in Section 5. In Section 6, we first describe how to derive error bounds on an operational model from a fully-parameterized one, and then we describe how this the operational model reveals errors in a published application note. We also mention future work.

2 Introduction to SAL

The protocols are specified and verified in the Symbolic Analysis Laboratory (SAL), developed by SRI, International [10]. SAL is a verification environment that includes symbolic and bounded model checkers, an interactive simulator, integrated decision procedures, and other tools.

SAL has a high-level modeling language for specifying transition systems. A transition system is specified by a *module*. A module consists of a set of state variables and guarded transitions. Of the enabled transitions, one is nondeterministically executed at a time. Modules can be composed both synchronously (`||`) and asynchronously (`[]`), and composed modules communicate via shared variables. In a synchronous composition, a transition from each module is simultaneously applied; a synchronous composition is deadlocked if either module has no enabled transition. In an asynchronous composition, an enabled transition from one of the modules is nondeterministically chosen to be applied.

The language is typed, and predicate sub-typing is possible. Types can be both interpreted and uninterpreted, and base types include the reals, naturals, and booleans; array types, inductive data-types, and tuple types can be defined. Both interpreted and uninterpreted constants and functions can be specified. This is significant to the power of these models: the parameterized values are uninterpreted constants from some parameterized type.

Bounded model checkers are usually used to find counterexamples, but they can also be used to prove invariants by induction over the state space [11]. SAL supports *k-induction*, a generalization of the induction principle, that can prove some invariants that may not be strictly inductive. By incorporating a *satisfiability modulo theories* decision procedure, SAL can do k-induction proofs over infinite-state transition systems.[4]

Let $(S, I, \rightarrow)$ be a transition system where S is a set of states, $I \subseteq S$ is a set of initial states, and $\rightarrow$ is a binary transition relation. If k is a natural number, then a *k-trajectory* is a sequence of states $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$ (a 0-trajectory is a single state). Let k be a natural number, and let P be property. The k-induction principle is then defined as follows:

– *Base Case*: Show that for each k-trajectory $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$ such that $s_0 \in I$, $P(s_j)$ holds, for $0 \leq j < k$.
– *Induction Step*: Show that for all k-trajectories $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$, if $P(s_j)$ holds for $0 \leq j < k$, then $P(s_k)$ holds.

The principle is equivalent to the usual transition-system induction principle when $k = 1$. In SAL, the user specifies the depth at which to attempt an induction proof, but the attempt itself is automated. The main mode of user-guidance in the proof process is in iteratively building up inductive invariants. While arbitrary LTL safety formulas can be verified in SAL using k-induction, only state predicates may be used as lemmas in a k-induction proof. Lemmas strengthen

[4] We use SRI's ICS decision procedure [12], the default SAT-solver and decision procedure in SAL, but others can be plugged in.

the invariant. We have more to say about the proof methodology for k-induction in Section 5.

3 Modeling and Verification of the Synchronizer Circuit

In this section we use a simple synchronizer circuit to illustrate the various modeling techniques used in this paper through the creation of successively more accurate models of the synchronizer circuit utilizing the transition language of SAL. In order to make the problem slightly more interesting, we generalize the data transfered by the circuit (`din`, `dout`) to arbitrary integers. Our initial model for the system of Figure 1 consists of two asynchronous processes – a transmitter (`tx`) and a receiver (`rx`).

```
system : MODULE = rx [] tx;
```

Thus, the transmitter and receiver execute in an interleaved fashion and at arbitrary rates; however, each is made up from several processes that are composed synchronously (i.e., operate in lock step). For example, the transmitter is composed of an "environment", which follows the basic protocol described above, and two instantiated flip-flops modules (described below) with their inputs and outputs suitably renamed.

```
tx : MODULE =  (   (RENAME d TO aout, q TO a1 IN FF)
                || (RENAME d TO a1, q TO ain IN FF)
                || tenv);
```

Our initial flip-flop model in Figure 4 has no provision for capturing timing constraints. Indeed, its behavior is simply an assignment that copies input `d` to output `q` without any reference to an underlying clock. Our models depend upon synchronous composition to force the flip-flops comprising the transmitter (and receiver) to execute in lock step.

```
FF : MODULE = BEGIN
                INPUT  d : BOOLEAN
                OUTPUT q : BOOLEAN
              INITIALIZATION
                q = FALSE
              TRANSITION
                q' = d
              END;
```

Fig. 4. Flip Flop

As mentioned, the transmitter's environment, shown in Figure 5, is constrained to obey the underlying protocol. There are two subtle points in this definition – we allow the data transmitted to take any randomly selected integer

value, and we allow the transmitter to "stutter" indefinitely when it is allowed to transmit a new value (stuttering is expressed by `guard -->` where `guard` is a boolean expression). The syntax `var IN range` defines a non-deterministic choice chosen from the set `range`. The infinite state model checker of SAL that enables our verification of timing constraints also enables verification with unbounded variables.

```
tenv : MODULE = BEGIN
          INPUT ain : BOOLEAN
          OUTPUT rout : BOOLEAN
          OUTPUT dout : INTEGER
       INITIALIZATION
          dout IN { x : INTEGER | TRUE };
          rout = FALSE
       TRANSITION
       [  TRUE        -->
       [] rout = ain  --> rout' = NOT rout;
                          dout' IN {x : INTEGER | TRUE };
       ] END;
```

Fig. 5. Transmitter's Environment

The receiver is similarly composed of an environment, flip-flops, and a data latch (the flipflop module in which the input and output variables are generalized to arbitrary integers).

```
rx : MODULE =
          ((RENAME d TO rout, q TO r1 IN FF)
        || (RENAME d TO r1, q TO rin IN FF)
        || (RENAME d TO dout, q TO din IN LATCH)
        || renv);
```

The receiver environment module non-deterministically stutters or echos `rin`.

```
renv : MODULE =
  BEGIN
    INPUT rin : BOOLEAN
    OUTPUT aout : BOOLEAN
  INITIALIZATION
    aout = FALSE
  TRANSITION
    aout' IN {aout, rin}
  END;
```

The defined circuit can be verified by induction over the (infinite) state space using the bounded model checking capabilities of SAL. In its current form, this circuit requires only straight induction ($k = 1$) for verification. Because the circuit implements a token passing protocol, a token counting lemma like the one in Figure 6 is key to its verification. Here, a "token" exists where the input and output to a flip-flop differ or where the receiver or transmitter environments are enabled to receive or send a value respectively; the syntax is the LTL temporal

logic where the G operator denotes that its argument holds in all states in a trajectory through the transition system. This lemma is used to prove the key theorem using simple induction:

```
changing(i : BOOLEAN, o : BOOLEAN) : [0..1] =
      IF (i /= o) THEN 1 ELSE 0 ENDIF;

l1 : LEMMA system |- G(changing(rin, r1) +
                         changing(r1, rout) +
                         changing(rout,ain) +
                         changing(ain, a1) +
                         changing(a1, aout) +
                         changing(aout,NOT rin)
                         <= 1);
```

Fig. 6. Counting Lemma

```
Sync_Thm : THEOREM system |- G((rout /= ain) => (dout = din));
```

Not surprisingly, both l1 and Sync_Thm can be verified quickly by SAL; however, the model as given does not capture any of the flip-flop timing requirements nor does it model any of the negative effects due to violating these requirements. In the following, we present a model that captures some of these requirements and allows us to verify the circuit even in the face of failures to meet these requirements.

We begin by modeling clocks. The transmitter and receiver are each composed with a local clock that regulates when that component may execute. The system we are developing has the following form:

```
(rx || rclock) [] (tx || tclock)
```

The basic idea, described as *timeout automata* by Dutertre and Sorea, is that the progress of time is enforced cooperatively (but nondeterministically) [13, 14]. The receiver and transmitter have *timeouts*, `rclk` and `tclk`, that mark the real-time at which they will respectively make transitions (timeouts are always in the future and may be updated nondeterministically). Each respective module representing the receiver and transmitter is allowed to execute only if its timeout equals the value of `time(rclk, tclk)`, which is defined to be the minimum of all timeouts.

```
time(t1 : TIME, t2: TIME): TIME =
     IF t1 <= t2 THEN t1 ELSE t2 ENDIF;
```

The receiver clock is defined in Figure 7. The transmitter clock is identical except for signal and constant names. As might be expected, the proof for the untimed model continues to work without change for this timed model since the addition of the timeout modules can only restrict the possible behaviors of

```
RPERIOD : { x : TIME | 0 < x};

rclock : MODULE = BEGIN
    INPUT  tclk : TIME
    OUTPUT rclk : TIME
    INITIALIZATION
        rclk IN { x : TIME | time(rclk,tclk) <= x }
    TRANSITION
        time(rclk,tclk) = rclk -->
           rclk ' IN { x : TIME | time(tclk,rclk) + RPERIOD <= x }
    END;
```

Fig. 7. Receiver Clock

the system and hence does not effect the safety property we are interested in verifying.

Our final refinement is to add a mechanism for defining timing constraints and for introducing behaviors that model the effect of violating these constraints. The approach we take is inspired by by a recent paper by Seshia et. al. describing the use of "Generalized Relative Timing "[15]. Briefly, we modify the described circuit elements to allow the aberrant behaviors that may arise due to violation of timing constraints and add "constraint" processes to regulate the conditions under which these aberrant behaviors may occur.

As mentioned previously, the behavior we wish to capture is due to metastability occurring when the inputs to a flip-flop do not satisfy timing requirements. The circuit design implicitly assumes that the period of the receiver and transmitter clocks are sufficiently long that metastability occurring at the beginning of a clock period will have been resolved prior to the next clock period. Thus, in the circuit described, the only signals which may exhibit metastability are `din`, `r1`, and `a1`. It is easy to demonstrate that the circuit will fail if this assumption is not met. Furthermore, the value of a signal after resolution of a metastable state is non-deterministic. We model this by replacing the key circuit elements with non-deterministic versions of the existing elements. For example, we define a non-deterministic flip-flop module in Figure 8. Similarly, we can define a non-deterministic latch which randomly selects its next output. The transmitter and receiver respectively are defined by appropriately renaming input and output variables, as shown in Figure 9.

Clearly, the circuit no longer satisfies its basic invariant. Our final step is to add processes that execute in parallel with the this system to constrain the outputs of the non-deterministic circuit elements. In particular, we assume that whenever `rout`, `aout`, or `dout` change state there is a settling period during which attempts to latch the new value will lead to metastability and hence a non-deterministic next state. As we shall show, the constraint processes that we add force the non-deterministic circuit elements to behave in a conventional manner outside these settling periods. The length of the settling period is implementation dependent and may be the result of a combination of factors such as signal propagation and circuit element setup time. In order to simplify the presentation,

```
FFnd : MODULE =
   BEGIN
     INPUT  d : BOOLEAN
     OUTPUT q : BOOLEAN
   INITIALIZATION
      q = FALSE
   TRANSITION
      q' IN {TRUE, FALSE}
   END;
```

Fig. 8. Nondeterministic Flip Flop

```
tx2 : MODULE =  (  (RENAME d TO aout, q TO a1 IN FFnd)
                || (RENAME d TO a1,  q TO ain IN FF)
                || tclock || tenv);

rx2 : MODULE =  (  (RENAME d TO rout, q TO r1 IN FFnd)
                || (RENAME d TO r1, q TO rin IN FF)
                || (RENAME d TO dout, q TO din IN LATCHnd)
                || rclock || renv );
```

Fig. 9. Transmitter and Receiver Modules

we have chosen to ignore hold time requirements. In practice, it is feasible to design flip-flops with zero-hold time requirements by inserting delays at the flip-flop input (at the cost of additional setup time). Furthermore, in an acyclic system such as 8N1 described in Section 4, one can simply shift the perspective of where the clock edge occurs to justify combining the setup and hold time requirements.

The system model, with the addition of the necessary constraints, has the form:

```
system : MODULE = (rx2 [] tx2) || constraints
```

Synchronous composition means that `rx2` and `tx2` can only execute when the necessary constraints are satisfied. Consider a flip-flop with input `d` and output `q`. We need a constraint module that monitors the `d` input for changes and constrains the `q` output to meet the requirements for "normal" behavior outside the settling period that follows a change, as shown in Figure 10 (note the module is a *parameterized module*; its parameter, `stime`, acts as a constant in the module). Consider the following constraint module, with appropriately renamed input and output variables.

```
(RENAME d TO rout, q to r1, dclk TO rclk, qclk TO tclk, ts TO r1ts IN
  Constraint[TSETTLE])
```

Whenever `rout` changes value and `rclk` preserves its value (i.e., `tx2` executes), the local timer `r1ts` is set to a value equal to the current time plus the settling constant `TSETTLE`. Whenever `rclk` changes value (i.e., `rx2` takes a step) either

```
Constraint [ stime : REAL ] : MODULE =
   BEGIN
     INPUT  dclk : TIME
     INPUT  qclk : TIME
     INPUT  d    : BOOLEAN
     INPUT  q    : BOOLEAN
     OUTPUT ts   : TIME
   INITIALIZATION
      ts = 0;
   TRANSITION
   [
      dclk /= dclk' AND (ts > time(dclk,qclk) OR q' = d) -->
   [] dclk = dclk' AND d /= d'                          --> ts' = time(dclk,qclk) + stime
   [] dclk = dclk' AND d = d'                           -->
   ]
END;
```

Fig. 10. Constraint Module

r1 is assigned rout or the local timer must be active. Finally, if neither condition occurs, the constraint module allows tx2 to execute. To constrain the three possible sources of non-deterministic behavior, there are three constraint modules with the local timers r1ts, a1ts, and d1ts monitoring changes on rout, aout, and dout, respectively.

The three constraint modules utilize two settling constants TSETTLE (for rout and dout) and RSETTLE (for aout). In verifying the circuit, we found that correct behavior depends on establishing a relationship between settling times and clock periods. In particular, the settling time of the transmitter must be less than the clock period of the receiver (and vice versa). Violating these assumptions has the effect of "injecting" additional tokens into the circuit whenever metastability occurs. Thus, we performed verification under the following assumptions.

```
TSETTLE  : { x : TIME | 0 <= x AND x < RPERIOD AND x < TPERIOD };
RSETTLE  : { x : TIME | 0 <= x AND x < RPERIOD AND x < TPERIOD };
```

With the changes described above, verification of the circuit behavior is more challenging, requiring k-induction over a modified token counting lemma and an additional helper lemma. To make the k-induction proof technique feasible, it is helpful to constrain the state space whenever possible. Hence, we developed the lemma shown in Figure 11 to assert certain obvious facts about system timing.

It was necessary to augment the counting lemma with additional addends to account for the possible spontaneous creation of tokens due to metastability, as shown in Figure 12.

Lemmas l0 and l1 can be proved at depth 1 (straight induction) with l1 using l0 as an assumption. The main theorem, Sync_Thm, can be verified at depth 3 using l0 and l1 as assumptions.

```
10 : LEMMA system |- G((r1ts <= time(rclk,tclk) OR
                           (r1ts + TPERIOD - TSETTLE <= tclk)) AND
                        (d1ts <= time(rclk,tclk) OR
                          (d1ts + TPERIOD - TSETTLE <= tclk)) AND
                        (a1ts <= time(rclk,tclk) OR
                        (a1ts + RPERIOD - RSETTLE <= rclk)) AND
                        (a1ts <= time(rclk,tclk) + RSETTLE) AND
                        (d1ts <= time(rclk,tclk) + TSETTLE) AND
                        (r1ts <= time(rclk,tclk) + TSETTLE ) AND
                        (time(rclk,tclk) <= rclk) AND
                        (time(rclk,tclk) <= tclk));
```

Fig. 11. Lemma 10

```
11 : LEMMA system |- G(changing(rout, r1) +
                      changing(r1, rin) +
                      changing(rin,aout) +
                      changing(aout, a1) +
                      changing(a1,ain) +
                      changing(ain,NOT rout) +
                      if (rout=r1 AND rclk < r1ts ) THEN
                          1 ELSE 0 ENDIF +
                      if (aout=a1 AND tclk < a1ts) THEN
                          1 ELSE 0 ENDIF
                     <= 1);
```

Fig. 12. Lemma 11

4 Modeling the 8N1 Protocol

In this section we discuss the model of the 8N1 protocol – its proof is deferred to Section 5. We model the protocol using two processes asynchronously composed – a transmitter (`tx`) and a receiver (`rx`). The general arrangement of the two major modules is illustrated in Figure 13. [5]

```
system : MODULE = rx [] tx;
```

As with the synchronizer circuit of Section 3, the transmitter and receiver each have a local clock module to manage their timeout. Recall that time is advanced whenever the module with the minimum timeout value executes and that the current time is always equal to the minimum timeout.

In addition to its local clock (`tclock`), the transmitter consists of an encoder (`tenc`) that implements the basic protocol, and an environment (`tenv`) that generates the data to be transmitted. These modules are synchronously composed.

```
tx : MODULE = tclock || tenc || tenv;
```

[5] Not shown are the shared variables used by the clock modules to compute the global "time".

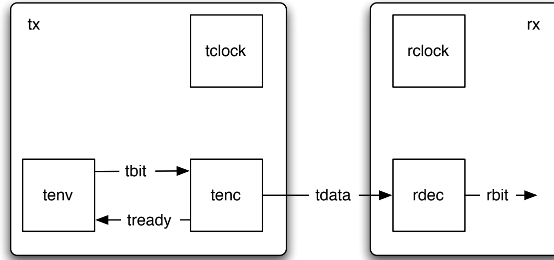

Fig. 13. System Block Diagram

Similarly, the receiver consists of its local clock (`rclock`) and and a decoder (`rdec`) that implements the protocol.

```
rx : MODULE = rdec || rclock;
```

The system is defined by the asynchronous composition of the transmitter and receiver which are then composed synchronously with a "constraint" module that models uncertainty in signal propagation as well as timing constraints. For the moment, we postpone discussion of the constraint module.

```
system : MODULE = (rx [] tx) || constraint;
```

The clock and environment modules for the transmitter are illustrated in Figure 14. The environment determines when new input data should be generated and is regulated by `tenc`. Whenever `tready` is true, a random boolean datum is selected; otherwise the old datum is preserved.

The timing model for the transmitter is similar to that for the synchronizer circuit. We assume an arbitrary clock period consisting of a settling phase (`TSETTLE`) and a stable phase (`TSTABLE`). The settling phase captures both setup requirements for the receiver as well as propagation delay. We will assume that reading the output of the transmitter `tdata` during the settling phase yields a non-deterministic result. As with the synchronizer, we assume that the receiver is implemented in such a manner that any metastability is resolved within the minimum clock period of the receiver. `TSETTLE` and `TSTABLE` are uninterpreted constants; however they are parameterized, which allows us to verify the model for any combination of settling time and receiver clock error (described subsequently). The transmitter settling time can be used to capture the effects of jitter and dispersion in data transmission as well as jitter in the transmitter's clock. In the case of the settling period, the model can be viewed as less deterministic than an actual implementation which might reach stable transmission values sooner. This means we verify the model under more pessimistic conditions than an actual implementation would face. As with the synchronization circuit, we do not actually model non-boolean values, rather we model a receiver that detects

```
TPERIOD : { x : TIME | 0 < x};
TSETTLE : { x : TIME | 0 <= x AND x < TPERIOD};

% function to compute current time

time(t1 : TIME, t2 : TIME) : TIME = IF t1 <= t2 THEN t1 ELSE t2 ENDIF;

tclock : MODULE =
 BEGIN
    INPUT  rclk   : TIME
    OUTPUT tclk   : TIME

 INITIALIZATION

    tclk  IN {x : TIME | 0 <= x AND x <= TSTABLE};

 TRANSITION
  [ tclk = time(tclk, rclk) -->  tclk' = tclk + TPERIOD;]
 END;

tenv : MODULE =
  BEGIN
    INPUT   tready : BOOLEAN
    OUTPUT  tbit   : BOOLEAN

  TRANSITION
   [
      tready --> tbit' IN {TRUE, FALSE}
   [] ELSE  -->
   ]
  END;
```

Fig. 14. Transmitter Environment and Clock

random values for signals that are not stable (as determined by the separate "constraint" module).

The transmitter encoder is defined as a simple state machine – state 0 corresponds to the start bit, states 1-8 correspond to the 8 data bit transmission states, and state 9 is the stop state. The encoder model is illustrated in Figure 15. Notice that the model allows the transmitter to stutter at state 9 indefinitely. The output `tdata` is either current value of `tbit` (states 1-8), FALSE (state 0), or TRUE (state 9).

The receiver clock is more complicated than the transmitter because of the manner in which a UART is implemented. Consider Figure 3. There may be an arbitrary "idle" period between frames during which the signal is high (TRUE). The behavior of a UART receiver is to "scan" for the high-to-low transition that marks the beginning of a frame. Once this transition is detected, the receiver predicts, based upon its local time reference, the middle of the 8 data and 1 stop bit times. There are two different intervals used for this prediction – the time between the detected "start" transition and the middle of the first data bit and the "period" between successive data samples. In an implementation, the bit period is generally an integer multiple of the scan time and the start interval is 1.5 times the bit period. Generally the bit time of the receiver is approximately that of the transmitter; however, in practice jitter and frequency errors mean

```
tenc : MODULE =
  BEGIN
    OUTPUT  tdata  : BOOLEAN
    OUTPUT  tstate : [0..9]
    OUTPUT  tready : BOOLEAN
    INPUT   tbit   : BOOLEAN
  INITIALIZATION
     tdata  = TRUE;
     tstate = 9;
  DEFINITION
     tready =  tstate < 8
  TRANSITION
   [
       tstate = 9 -->
    [] tstate = 9 --> tdata' = FALSE;
                      tstate' = 0;
    [] tstate < 9 --> tdata' =  (tbit' OR tstate = 8);
                      tstate' = tstate + 1;
   ]
  END;
```

Fig. 15. Transmitter Encoder

that each measurement interval is subject to error. In our model we associate all errors with the receiver and assume that the transmitter runs at a constant rate.

The various receiver clock periods are expressed in terms of linear equations that define lower and upper bounds for “SCAN”, “START”, and “PERIOD”. The details of these equations can be viewed as part of the proof – we verify the protocol subject to these bounds – and are postponed to Section 5. The receiver clock along with the various is illustrated in Figure 16. The specific timeout interval depends upon the state of the decoder; i.e., whether the decoder is scanning, sampling the first data bit, or sampling subsequent data bits.

The decoder is illustrated in Figure 17. There are three transitions – the first two model the non-deterministic choice that occurs when scanning for the start bit and a third models sampling the data bits. The receiver has 10 states (numbered [0..9]) where the 8 data bits are received in states 0-7, the stop bit is received in state 8, and scanning for a new start bit occurs in state 9.

As with the synchronizer, the value of the bit read is always chosen non-deterministically, though the next state may depend upon the specific choice. Furthermore, the choice is constrained by a separate module that determines when the sampled value should reflect the input (`tdata`) and when the sampled value may be random. The constraint module is also presented in Figure 17. The only significant difference between this and the constraint modules used with the synchronizer is the extra output `stable` which is used in developing the proof.

```
timeout (min : TIME, max : TIME) : [TIME -> BOOLEAN]  =
              { x : TIME | min <= x AND x <=  max};

rclock : MODULE  =
  BEGIN
    INPUT tclk   : TIME
    INPUT rstate : [0..9]
    OUTPUT rclk  : TIME
  INITIALIZATION
    rclk IN { x : TIME | 0 <= x AND x < RSCANMAX };
  TRANSITION
   [
    rclk = time(rclk, tclk) -->
         rclk' IN IF (rstate' = 9) THEN
                    timeout(rclk + RSCANMIN, rclk + RSCANMAX)
                  ELSIF (rstate' = 0) THEN
                     timeout(rclk + RSTARTMIN, rclk + RSTARTMAX)
                  ELSE
                     timeout(rclk + RPERIODMIN, rclk + RPERIODMAX)
                  ENDIF;
   ]
   END;
```

Fig. 16. Receiver Clock

5 Verification of the 8N1 Protocol

Our main goal is to prove that the 8N1 decoder reliably extracts the data from the signal it receives.

```
Uart_Thm :  THEOREM system |- G(rstate < 9 AND
                                rstate > 0 AND
                                rclk >= tclk => ((tstate = rstate)  AND
                                                 (rbit = tbit)));
```

Briefly, the theorem states that immediately after the receiver executes each of its 8 bit receive states (0..7), the received bit is equal to the currently transmitted bit. This interpretation of the theorem depends upon the knowledge that the states of the transmitter and receiver obey the following sequence. This sequence is verified with theorem t0 to be discussed subsequently.

$$(tstate, rstate) = (9, 9), (0, 9), (0, 0), (1, 0), (1, 1), ...(9, 8), (9, 9) \quad (2)$$

As mentioned previously, an important component of the proof is the set of bounds on the various time constants utilized in the decoder model. We derived the bounds by assuming worst case (minimum or maximum) and then determining how temporal errors accumulate by the 10th bit time (the stop bit). Informally, the correct behavior of the protocol requires that all samples other than the initial scan fall during the "stable" portion of the transmitter clock. We derived these bounds by considering the execution sequence described and with the knowledge that the correct behavior of the receiver requires that in receiver states 0..8, we require the clock events fall during the "stable" period

```
rdec : MODULE =
  BEGIN
    INPUT  tdata  : BOOLEAN
    OUTPUT rstate : [0..9]
    OUTPUT rbit   : BOOLEAN
  INITIALIZATION
    rbit  = TRUE;
    rstate = 9;
  TRANSITION
   [
      rstate = 9  -->  rbit' = TRUE
   [] rstate = 9  -->  rbit' = FALSE;
                        rstate' = 0
   [] rstate /= 9 -->  rbit' IN {FALSE, TRUE};
                        rstate' = rstate + 1
   ]
  END;

constraint : MODULE =
  BEGIN
    INPUT  tclk    : TIME
    INPUT  rclk    : TIME
    INPUT  rbit    : BOOLEAN
    INPUT  tdata   : BOOLEAN
    OUTPUT stable  : BOOLEAN
    LOCAL changing : BOOLEAN
  DEFINITION
    stable = (NOT changing OR (tclk - rclk < TSTABLE));
  INITIALIZATION
    changing = FALSE
  TRANSITION
   [
      rclk' /= rclk AND (stable => rbit' = tdata) -->
   [] tclk' /= tclk --> changing' = (tdata' /= tdata)
   ]
  END;
```

Fig. 17. Receiver Decoder and Constraint Module

of the transmitter. Consider the case of the "scan" operation. In order to detect the start bit, we must guarantee that the receiver sample `tdata` with a period that is no longer that the stable period – if the interval were longer, then the start bit might might be missed because two successive samples by the receiver fall outside the stable interval.

```
RSCANMIN : { x : TIME | 0 < x };
RSCANMAX : { x : TIME | RSCANMIN <= x AND x < TSTABLE };
```

Once the start bit is detected, the receiver waits for a "start" time before reading the first data bit. Reading this data bit must fall in the stable region for transmitter state 1.

```
RSTARTMIN : { x : TIME | TPERIOD + TSETTLE < x };
RSTARTMAX : { x : TIME | RSTARTMIN <= x AND
                 TSETTLE + RSCANMAX + x < 2 * TPERIOD };
```

In subsequent states the receiver clock error accumulates. Thus, the constraint on the receiver "period" depends upon the accumulated error at the point of sampling the stop bit.

```
RPERIODMIN : { x : TIME | 9 * TPERIOD + TSETTLE < RSTARTMIN + 8 * x };
RPERIODMAX : { x : TIME | RPERIODMIN <= x AND
                 TSETTLE + RSCANMAX + RSTARTMAX + 8 * x < 10 * TPERIOD };
```

The proofs of t0 and Uart_Thm require supporting lemmas. In general, when a k-induction proof attempt fails, two options are available to the user: the proof can be attempted at a greater depth, or supporting lemmas can be added to restrict the state-space. A k-induction proof attempt is automated, but if the attempt is not successful for a sufficiently small k (i.e., the attempt takes too long or too much memory), additional invariants are necessary to reduce the necessary proof depth. The user must formulate the supporting invariants manually, but their construction is facilitated by the counterexamples returned by SAL for failed proof attempts. If the property is indeed invariant, the counterexample is a trajectory that fails the induction step but lies outside the set of reachable states, and the state-space can be appropriately constrained by an auxiliary lemma based on the counterexample. The following lemmas are built by examining the counterexamples returned from proof attempts for the main theorem and the successive intermediary lemmas.

Once it is determined what property the states fail to have that makes them unreachable, this property can be stated (and proved) as an additional predicate. This predicate is used as a lemma to support the proof original of the original property. The following lemmas capture some simple facts about the relationships between the two clocks. Of these, l1, is the least obvious and was derived along with theorem t0 in order to reduce the required induction depth. Each of these lemmas is inductive and hence can be proved at depth 1.

```
l1 : LEMMA system |- G(tclk <= (rclk + TPERIOD) OR stable);

l2 : LEMMA system |- G(rclk <= tclk + RSTARTMAX OR
                       rclk <= tclk + RSCANMAX OR
                       rclk <= tclk + RPERIODMAX);
```

The key part of our proof of 8N1 is an invariant that describes the relationship between the transmitter and receiver. We must relate them both temporally and with respect to their discrete state (e.g., `tstate` with `rstate` and `tdata` with `rbit`). The number of and the complexity of the supporting lemmas necessary to prove the main results is significantly reduced by proving a *disjunctive invariant* [16]. A disjunctive invariant has the form $\bigvee_{i \in I} P_i$ where each P_i is a state predicate (predicates P_i and P_j need not be disjoint for $i \neq j$). Disjunctive invariants are easier to generate iteratively than conjunctive invariants. If a disjunctive invariant fails to cover the reachable states, additional disjuncts can be incrementally added to it (in a conjunctive invariant, additional conjunctions must hold in all the reachable states). Although this is a general proof technique,

it is particularly easy to build a disjunctive invariant in SAL. The counterexamples SAL returns can be used to iteratively weaken the disjunction until it is invariant.

Theorem t0 has 20 disjuncts corresponding to the 20 unique states in equation 2. Of the disjuncts, 18 follow a simple pattern and are defined in SAL with a recursive function. The following defines theorem t0.

```
t0 : THEOREM system |- G(
    % idle
     ((rstate =  9) AND
      (tstate = 9) AND
      (tdata AND rbit) AND
      stable AND
      (rclk - tclk <= RSCANMAX))
 OR % start bit sent, not detected
     ((rstate = 9) AND
      (tstate = 0) AND
      (NOT tdata AND rbit) AND
      (rclk - tclk <= RSCANMAX - TSTABLE))
 OR % --- unwind all the other cases
    rec_states(8, tstate, rstate, tdata, rbit, rclk, tclk, stable));
```

The recursively defined disjuncts use the following pattern for $n = 0..8$.

```
  ((tstate = n + 1) AND
   (rstate = n) AND
   (rclk - tclk <=
      mult(n, RPERIODMAX) - mult(n+1, TPERIOD) + RMAX - TSTABLE) AND
   (rclk - tclk >=
      mult(n, RPERIODMIN) - mult(n+1, TPERIOD) + RSAMPMIN - TPERIOD))
 OR
  ((tstate = n) AND
    (rstate = n) AND
     stable AND
    (tdata = rbit) AND
    (rclk - tclk <=
       mult(n, RPERIODMAX) - mult(n, TPERIOD) + RMAX - TSTABLE) AND
    (rclk - tclk >=
       mult(n, RPERIODMIN) - mult(n+1, TPERIOD) + RSAMPMIN));
```

In general, each disjunct defines the control state (`tstate` and `rstate`), the constraints on the data signals if any, and describes the relative difference between `tclk` and `rclk`. A bug in ICS which involved multiplication of uninterpreted constants required a work-around in which we defined multiplication recursively. This theorem can be proved at depth 3, while the main theorem (Uart_Thm) can then be proved at depth 2 with t0 as a lemma.

6 Discussion

Our proof of the 8N1 protocol is verified with respect to bounds on the various timing constants. In a practical implementation, the receiver scan period is defined relative to the nominal transmitter bit period and the receiver start and bit periods are integer multiples of this. What an implementor ultimately cares about is the the trade off between settling time (in general due to signal dispersion over a given transmission medium) and frequency error.

In the following, we show how the bounds that we have verified can be used to derive error and settling time bounds in a form that is more convenient for a protocol implementer. These derived bounds are somewhat more restrictive than what we have verified since we require the maximum allowable frequency error to be symmetric about the nominal frequency. As before, let `TPERIOD` be the nominal period duration. We introduce another uninterpreted constant in the operational model representing the nominal duration the receiver waits for the start bit ("START").

```
TSTART : TIME;
```

```
RSTARTMAX : TIME = TSTART * (1 + ERROR);
RSTARTMIN : TIME = TSTART * (1 - ERROR);
RSCANMAX : TIME = 1 + ERROR;
RSCANMIN : TIME = 1 - ERROR;
RPERIODMAX : TIME = TPERIOD * (1 + ERROR);
RPERIODMIN : TIME = TPERIOD * (1 - ERROR);
```

Fig. 18. Receiver Parameters Defined with respect to Error

Now, let `ERROR` be an uninterpreted constant from `TIME`, and then the constants in Figure 10 are defined in terms of `ERROR`. By replacing these defined terms in the parameterization of the types in Sec 5, we compute the bound on the error. For example, `RSTARTMAX` is an uninterpreted constant from the following parameterized type:

```
RSTARTMAX : { x : TIME | RSTARTMIN <= x AND
                         TSETTLE + RSCANMAX + x < 2 * TPERIOD };
```

Replacing `RSTARTMIN` and `RSCANMAX` by their definitions from Figure 18, we get

```
RSTARTMAX : { x : TIME | TSTART * (1 - ERROR) <= x AND
                         TSETTLE + 1 + ERROR + x < 2 * TPERIOD };
```

By replacing each term with its definition, the type parameters are defined completely in terms of `TPERIOD`, `TSETTLE`, and `ERROR`. Isolating `ERROR` in the system of inequalities gives bounds on `ERROR`. For the 8N1 protocol, `ERROR` is thus parameterized as follows:

```
ERROR : { x : TIME | 0 <= x AND
          (9 * TPERIOD + TSETTLE <
              8 * TPERIOD * (1-x) + TSTART * (1-x)) AND
          ((8 * TPERIOD * (1+x) + TSTART * (1+x) + (1+x) + TSETTLE) <
              10 * TPERIOD) };
```

This derived model can be verified using the same invariants proved at the same depth as in the verification described in Section 5.

As mentioned in Section 1, we discovered significant errors in the analysis in an application note for UARTs [2]. For `TPERIOD = 16` and `TSTART = 23`, the authors suggest that if `TSTABLE` is `TPERIOD/2` (they call this the "nasty" scenario), then a frequency error of $\pm 2\%$ is permissible. In fact, even with zero frequency mismatch, the stable period is too short – if we assume "infinitely" fast sampling, it is possible to show that the settling time must be less than 50% of `TPERIOD`. In other words, the type parameterizing `ERROR` is empty when `TSTABLE` is `TPERIOD/2` (this can be shown using SAL or by a simple calculation). With our choice of time constants, the longest settling time must be less than 7 (43.75%). In reading the article, it becomes clear that the authors neglected the temporal error introduced by sampling the start bit. They describe a "normal" scenario with `TSETTLE = TPERIOD/4` and assert that a frequency error of $\pm 3.3\%$ is permissible. As our derivation above illustrates, the frequency error in this case is limited to $\pm 3/151 \approx \pm 1.9\%$.

This paper describes the use of SAL to model and verify a data synchronization circuit and the 8N1 protocol. We show, by example, how models of these can be refined in the language of SAL to capture timing constraints and environmental effects such as metastability and settling. Future work includes extending this framework to other cross domain protocols as well as developing the theory for refinement.

Acknowledgments

We thank Leonardo de Moura, John Rushby, and anonymous reviewers for a recent paper [17] for their suggestions and corrections.

References

1. 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.
2. Maxim Integrated Products, Inc. *Determining Clock Accuracy Requirements for UART Communications*, June 2003. Available at `http://www.maxim-ic.com/appnotes.cfm/appnote_number/2141`.
3. Tsachy Kapschitz and Ran Ginosar. Formal verification of synchronizers. In *CHARME 2005 – to appear*, 2005.
4. Tsacky Kapschitz, Ran Ginosar, and Richard Newton. Verifying synchronization in multi-clock domain SoC. In *DVCon 2004*, 2004.

5. Tai Ly, Neil Hand, and Chris Ka-Kei Kwok. Formally verifiying clock domain crossing jitter using assertion-based verification. In *DVCon 2004*, 2004.
6. Karen Yorav, Sagi Katz, and Ron Kiper. Reproducing synchronization bugs with model checking. In *CHARME*, pages 98–103, 2001.
7. T. Henzinger, J. Preussig, and H. Wong-Toi. Some lessons from the Hytech experience. In *Proceedings of the 40th Annual Conference on Decision and Control*, pages 2887–2892, 2001.
8. Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. TReX: A tool for reachability analysis of complex systems. In *Computer-Aided Verification, CAV'01*, pages 368–372, London, UK, 2001. Springer-Verlag.
9. F. W. Vaandrager and A. L. de Groot. Analysis of a biphase mark protocol with Uppaal and PVS. Technical Report NIII-R0445, Radboud University Nijmegen, 2004.
10. Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *Computer-Aided Verification, CAV'04*, volume 3114 of *LNCS*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.
11. Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In *Computer-Aided Verification, CAV'03*, volume 2725 of *LNCS*, 2003.
12. Leonardo de Moura, Sam Owre, Harald Ruess, John Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In *2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *LNCS*, pages 218–222, Cork, Ireland, July 2004. Springer-Verlag.
13. Bruno Dutertre and Maria Sorea. Timed systems in SAL. Technical Report SRI-SDL-04-03, SRI International, 2004.
14. Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *FORMATS/FTRTFT*, pages 199–214, 2004.
15. Sanjit A. Seshia, Randal E. Bryant, and Kenneth S. Stevens. Modeling and verifying circuits using generalized relative timing. In *ASYNC*, pages 98–108, 2005.
16. John Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In *Computer-Aided Verification, CAV'00*, volume 1855 of *LNCS*, pages 508–520, Chicago, IL, July 2000. Springer-Verlag.
17. Geoffrey M. Brown and Lee Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, 2006. To appear. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/bmp.html`.