DOI 10.1007/s00165-006-0008-1
BCS © 2006
Formal Aspects of Computing (2006) 18: 433–458

Formal Aspects
of Computing

Analysis of a biphase mark protocol with UPPAAL and PVS

F.W. Vaandrager[1] and A.L. de Groot[2]

Institute for Computing and Information Sciences, Radboud University Nijmegen, Nijmegen, The Netherlands.
E-mail: F.Vaandrager@cs.ru.nl; A.deGroot@cs.ru.nl

Abstract. The biphase mark protocol is a convention for representing both a string of bits and clock edges in a square wave. The protocol is frequently used for communication at the physical level of the ISO/OSI hierarchy, and is implemented on microcontrollers such as the Intel 82530 Serial Communications Controller. An important property of the protocol is that bit strings of arbitrary length can be transmitted reliably, despite differences in the clock rates of sender and receiver (drift), variations of the clock rates (jitter), and distortion of the signal after generation of an edge. In this article, we show how the protocol can be modelled naturally in terms of timed automata. We use the model checker UPPAAL to derive the maximal tolerances on the clock rates, for different instances of the protocol, and to support the general parametric verification that we formalized using the proof assistant PVS. Based on the derived parameter constraints we propose instances of BMP that are correct (at least in our model) but have a faster bit rate than the instances that are commonly implemented in hardware.

Keywords: Biphase mark protocol; Formal methods; Model checking; Theorem provers; Timed automata

1. Introduction

The biphase mark protocol (BMP) is a fundamental protocol that is widely used in applications where data written by one device is read by another. It is used, for instance, in microcontrollers such as the Intel 82530 Serial Communications Controller [Int91], in some optical communications and satellite telemetry applications, and for communication between consumer electronics devices. A variation of biphase mark, called "Manchester", is used in the Ethernet. The first rigorous, formal analysis of (some instances of) the protocol was carried out by Moore [Moo94] using the Boyer-Moore theorem prover Nqthm [BM88]. Moore used the biphase mark protocol to illustrate a formal, logical model of asynchronous communication. We refer to [Moo94] for additional information and references on BMP.

In this article, we present the results of our efforts to model and analyze the biphase mark protocol using the verification tools UPPAAL and PVS. Our model generalizes Moore's model, since it incorporates "clock jitter" and the distortion in the signal due to the presence of an edge is not limited to the time-span of the cycle during which the edge was written. We use UPPAAL [LPY97], a model checker for networks of timed automata, to automatically prove correctness of several instances of the protocol and to find error scenarios based on some

Correspondence and offprint requests to: F.W. Vaandrager, E-mail: F.Vaandrager@cs.ru.nl

[1] F.W. Vaandrager was supported by EU IST project IST-2001-35304 Advanced Methods for Timed Systems (AMETIST).

[2] A.L. de Groot was supported by NWO project 612.062.000 Architecture for Structuring the requirements Specification of Embedded Safety-critical Systems (ASSESS).

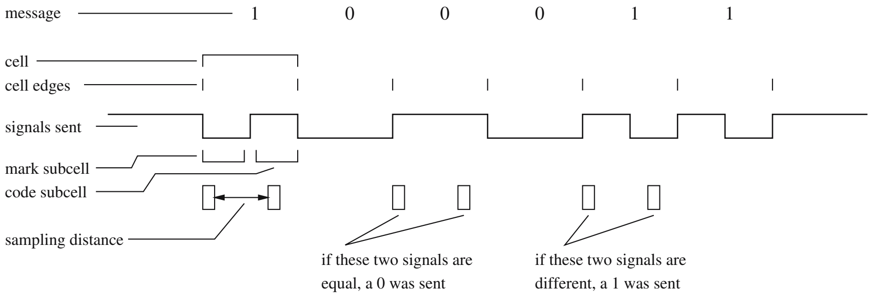
message
1
0
0
0
1
1
cell
cell edges
signals sent
mark subcell
code subcell
sampling distance
if these two signals are equal, a 0 was sent
if these two signals are different, a 1 was sent

Fig. 1. Biphase mark terminology

incorrect instances. These experiments suggest constraints on the model parameters that are necessary for correctness. Using the proof assistant PVS [ORSH95] we establish that these constraints are in fact sufficient for correctness. Our main objective for this article is to demonstrate that the timed automata framework [AD94] allows for natural, straightforward modeling and analysis of this type of physical level communications protocols. Methodologically, our results are interesting since we use two different tools – the model checker UPPAAL and the proof assistant PVS – in a combined manner to analyze a single system. Both tools play a vital and complementary role in our analysis. As a byproduct of our efforts, we manage to find instances of BMP that are correct (at least in our model) but have a faster bit rate than the instances that are commonly implemented in hardware.

Outline. Section 2 contains an informal presentation of the protocol. In Sect. 3 we present our UPPAAL model and the parameter constraints that we inferred by trying to verify several instances of the protocol with UPPAAL model checker. Section 4 describes a straightforward encoding of the (semantics of) the UPPAAL model within the higher order logic input language of the proof assistant PVS. Section 5 reports on the formalization with PVS of the (manual) proof that the parameter constraints are sufficient for correctness. In Sect. 6, we investigate the consequences of the derived parameter constraints. In particular, we show how to syntesize the instance of BMP with the fastest bit rate given an upper bound on the time needed for the signal to stabilize after occurrence of an edge. Finally, Sect. 7 discusses related work and draws some conclusions.

The full UPPAAL and PVS sources are available at the URL
http://www.cs.ru.nl/ita/publications/papers/fvaan/BMP.html.

2. Informal description of the protocol

Essentially, the biphase mark protocol is just a convention for representing both a string of bits and clock edges in a square wave. In the protocol (see Fig. 1, taken from [Moo94]) each bit of a message is encoded in a *cell*, which consists of a number of clock cycles and which is logically divided into a *mark subcell* and a *code subcell*. A typical configuration is 16 cycles for the mark subcell and another 16 for the code subcell. The signal, which at any time is either *high* or *low*, always changes value right at the beginning of a cell. In addition, if the signal encodes a "1" the value also changes right at the beginning of the code subcell. If a cell encodes a "0" then the signal remains constant throughout the cell. In order to decode the bit string again from the square wave, a decoder waits for an edge that marks the beginning of a cell and then samples the wire after a specified number of clock cycles (*the sampling distance*). If the value just after the edge agrees with the sampled value then the decoder assumes a "0" has been sent, otherwise it assumes a "1" has been sent.

A clear advantage of BMP over, say, the naive scheme in which a "1" is encoded by a high signal and a "0" by a low signal, is that BMP "synchronizes" the clocks of coder and decoder at the beginning of each cell. In a setting with unreliable clocks, a decoder might not see the difference between the naive encoding of 10,000 consecutive 1's, and the naive encoding of 10,001 consecutive 1's. If BMP is used this problem does not arise, except of course when clocks become excessively (depending on the other parameters of the protocol, usually 4 or 5 orders of magnitude worse than common clock crystals provide) unreliable.

Proving correctness of physical implementations of BMP is nontrivial due the combination of at least three factors:

1. A physical system can not generate a perfect square wave. Edges will not be vertical or square: an electrical signal changes continuously and may even "ring" before stabilizing at its new level. In our model we will

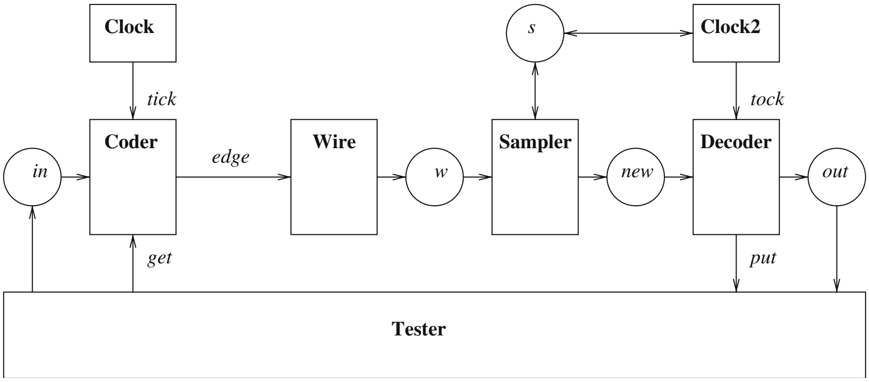

Fig. 2. Architecture of the UPPAAL model

assume that the value of the signal is nondeterministically defined (reading may produce any value) during some bounded interval after the coder generates an edge.

2. If a signal is constant throughout a clock cycle then we may assume that sampling of this signal by the decoder yields the correct value. However, if the value changes during a clock cycle then any value may come out as we do not know at which point during the cycle sampling takes place. In our model we will assume that the decoder samples the signal nondeterministically at some point during each clock cyle.
3. Physical clocks drift (that is, their rate may be too high or too low) and exhibit jitter (that is, their rate may change over time). In our model we will assume that the clock rates of sender and receiver are contained in an interval, so that subsequent clock ticks may be separated by *any* length of time in that interval.

As a consequence of these complications, one can easily imagine scenarios in which, for instance, a decoder altogether misses an edge and completely garbles the remainder of the signal (in Sect. 3.10 we will present such scenarios). In this article we identify the precise constraints on the various parameters of the protocol (lengths of clock cycles, time before signal stabilizes, et cetera) that must be met in order to ensure correctness.

3. UPPAAL model and analysis

In this section, we describe the model that we constructed of the biphase mark protocol and the analysis results that we obtained using the timed model checking tool UPPAAL. For a detailed account of UPPAAL we refer to [LPY97] and to http://www.uppaal.com.

3.1. Architecture

Figure 2 presents the overall architecture of our UPPAAL model. The model consists of a network of 7 timed automata (shown as rectangles), which communicate via shared variables (circles) and synchronization actions (labeled arrows). Automaton **Clock** models the hardware clock at the coding side. The automaton **Coder** models the encoding process: based on a sequence of bits (which is received via variable *in*) and the *tick* events from the **Clock** automaton, it generates *edge* events that determine a square wave. Within our model, the environment (which is represented by the tester) places a new bit in variable *in* whenever the **Coder** is willing to accept a *get* event. The **Wire** automaton nondeterministically transforms the perfect square wave from the **Coder** into a signal whose value, stored in variable w, is nondeterministically defined during a specified interval after the coder generates an edge. Automaton **Clock2**, which is similar to **Clock**, models the hardware clock at the decoding side. The **Sampler** automaton periodically copies (samples) the value of variable w into variable *new*. The Boolean variable s is used to coordinate the sampler and clock. Automaton **Decoder** models the decoding process. If at the occurrence of a clock tick automaton **Decoder** observes that the value of *new* has changed it starts counting a specified number of clock ticks and then compares the value of *new* after those ticks with the value it had before. Depending on the outcome it places either a 0 or a 1 in register *out* and informs the environment about the fact that a new bit has become available via a *put* action. The automaton **Tester**, finally, nondeterministically selects bits, places them in register *in* upon request and checks whether the sequence of bits delivered via register *out*

cell	16
mark	8
sample	11
min	89
max	100
edgelength	89

Fig. 3. Parameters of the UPPAAL model

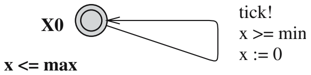

Fig. 4. Clock

agrees with the sequence entered via register *in*. Whenever it observes a discrepancy, the **Tester** automaton jumps to a designated error location. Hence, in order to establish correctness we must prove that the error location can not be reached.

3.2. Model parameters

Figure 3 lists the parameters that are used in the model (constants in UPPAAL terminology) and gives an example instantiation. The domain of all parameters is the set of natural numbers. Constant cell specifies the size of a cell in terms of the number of clock cycles. Similarly, mark and sample specify the size of the mark subcell and the sampling distance, respectively. This specific configuration is used in the Intel 82530 Serial Communications Controller [Int91]. Constants min and max specify the minimum and maximum number of time units in a clock cycle (say, measured in nanoseconds); we assume $0 < \mathsf{min} \leqslant \mathsf{max}$. Constant edgelength specifies the number of time units needed for the signal to stabilize after occurrence of an edge. The values listed for min, max and edgelength are not meant to be realistic – our model's clocks are much worse than any that are used in real machines [Cri89].

3.3. First clock

Timed automaton **Clock** models the hardware clock at the coding side. The automaton, which is displayed in Fig. 4, only has a single location and a single transition. The automaton performs a synchronization action *tick*! when its clock x has reached a value between min and max, and then returns to its initial state by resetting x.

3.4. The coder

We worked hard to make all the timed automata as simple as possible. As a result of our efforts, all automata in our model have at most five locations.[3] The automaton **Coder**, displayed in Fig. 5, is one of the two timed automata in our model with five locations. The automaton **Coder** describes how the biphase mark protocol encodes a string of bits and clock edges into a square wave. In its initial location **C0** the automaton immediately (the location is urgent) jumps via a *get*? transition to location **C1**, thereby telling the environment that it is about to fetch a new bit from the *in* register. In the location **C1**, which is also urgent, an edge is generated and the automaton jumps, depending on the bit that is being transmitted, either to location **C2** (in case $in = 1$) or to location **C3** (in case $in = 0$). A local integer counter n is used to count clock ticks. Upon entering location **C2** the automaton waits until mark clock ticks have occurred, and then generates an edge and jumps to location **C3**. In location **C3** the automaton waits until cell clock ticks have occurred and then jumps back to its initial location to transmit the

[3] Actually, the number of locations is not a good measure of complexity since in the presence of integer variables each timed automaton is trivially equivalent to one with just a single location. Without introducing any additional integer variables or coding tricks we could have easily reduced the number of locations of the **Coder** automaton to 3 if UPPAAL would have permitted us to decorate transitions with multiple synchronization labels, as in the tool Kronos [BDM+98].

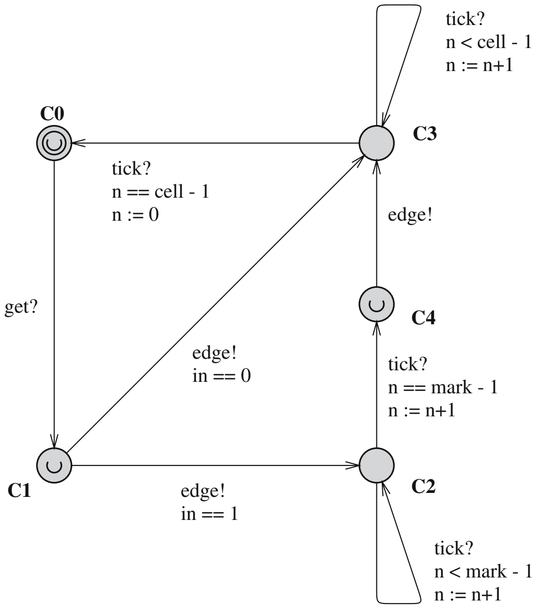

Fig. 5. Coder

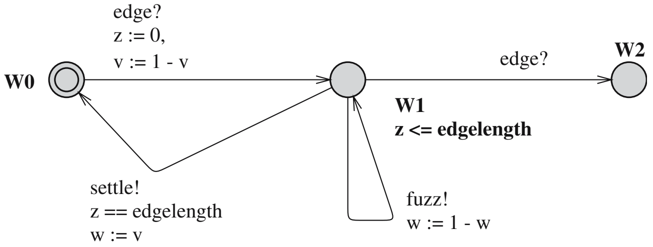

Fig. 6. Wire

next bit. In our model, we assume that there is always a next bit to transmit. It should not be difficult to generalize our work to a setting where sometimes the environment has no more bits available for transmission.

3.5. The wire

The **Wire** automaton, displayed in Fig. 6, is introduced to model our assumption that it takes edgelength time before an electric signal stabilizes after occurrence of an edge. The Boolean variable v is toggled when an *edge*? event occurs. Thus, v evolves according to the perfect square wave that is generated by the **Coder**. There is also another Boolean variable w, whose values reflect the actual observations that can be made on a physical wire. In the initial location **W0** the wire is stable and the values of v and w agree. Upon occurrence of an *edge*? the **Wire** automaton moves to the unstable location **W1** in which w can be assigned any value at any time. After being unstable for edgelength time units, the system moves back again to the stable location **W0** and the value of w settles to v. For the parameter assignments for which the BMP is correct the **Coder** never generates an *edge* if the **Wire** is in location **W1**. We will prove this by establishing that location **W2** is unreachable in the full system for any of the parameter assignments that we consider.

We find it convenient to give names to all the transitions in an automaton. This is achieved by misusing the broadcast primitive in UPPAAL: the broadcast actions *fuzz*! and *settle*! do not synchronize with actions from any other automaton, but are just there to give transitions a name.

In our model we assume instantaneous message delivery: edges generated by the **Coder** may be detected instantaneously by the **Decoder**. We claim that the constraints on the parameters that we derive in this paper to

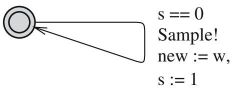

Fig. 7. Sampler

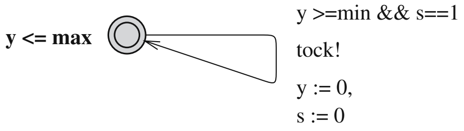

Fig. 8. Clock2

ensure correctness are not affected when we introduce a fixed transmission delay for all edges. However, we do not formally prove this claim in this article.

3.6. The sampler

The **Sampler** automaton, displayed in Fig. 7, only has a single location and a single transition. The transition copies (samples) the value of the wire variable w to a variable *new* that is used as input for the decoder. To ensure that the sample transition occurs exactly once during every clock cycle we use an auxiliary Boolean variable s: if $s = 0$ then the sampler may sample and if $s = 1$ then the (decoder) clock may tick. Only the samples taken during the mark and sample clock cycles are actually used in the protocol, though.

3.7. Second clock

The **Clock2** automaton, displayed in Fig. 8, models the hardware clock at the decoding side. This automaton is exactly the same as the **Clock** at the coder side, except that it also reads/writes variable s to ensure strict alternation of the *sample* and *tock* actions.

3.8. The decoder

The **Decoder** automaton, shown in Fig. 9, models in a straightforward manner the decoding of the (sampled) wire signal into a bit string. The automaton uses a local Boolean variable *old* to record wire values it has seen earlier. Like the **Coder** automaton, the activity of the **Decoder** automaton is driven by clock ticks. In the initial location **D0**, each clock tick causes the automaton to compare the most recent value that has been sampled from the wire (*new*) with the value stored in *old*. As long as these values remain the same no action is taken. But as soon as the values of *old* and *new* become different, the automaton concludes that an edge has occurred, moves to location **D1**, and toggles the value of *old*. In location **D1** the automaton waits until sample clock ticks have occurred (counted using a local integer variable m) and then jumps to location **D2**. If at that point in time *new* equals *old* then output variable *out* is assigned the value 0, otherwise *out* is assigned the value 1 (cf Fig. 1). In a subsequent transition that occurs immediately, the environment is informed that a new output has been produced (via a *put*! synchronization) and the **Decoder** returns to its initial location.

3.9. The tester

Figure 10 depicts the **Tester** automaton, the seventh and last component of the model. This automaton is not part of the biphase mark protocol but just a highly nondeterministic environment of the protocol that has been designed to test its correctness. If the protocol (the **Coder** in fact) asks for a new bit, the **Tester** puts a nondeterministically selected bit in shared variable *in*. The **Tester** remembers which bits it has sent to the protocol; *in* and *buf* store the most-recent two bits sent. If a third bit is requested by the **Coder** the overflow location **T3** is reached We will prove that for all parameter assignments for which the protocol operates correctly, there is at most one

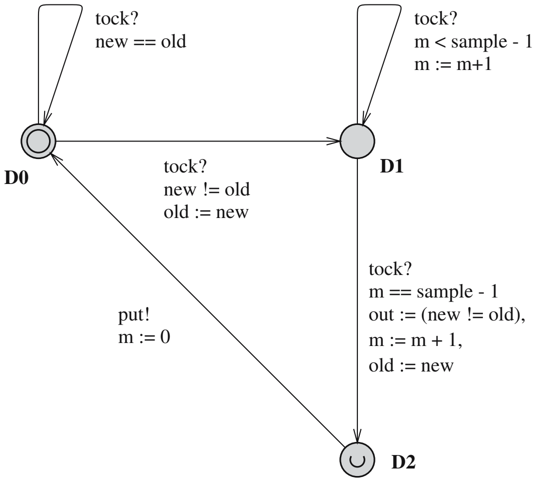

Fig. 9. Decoder

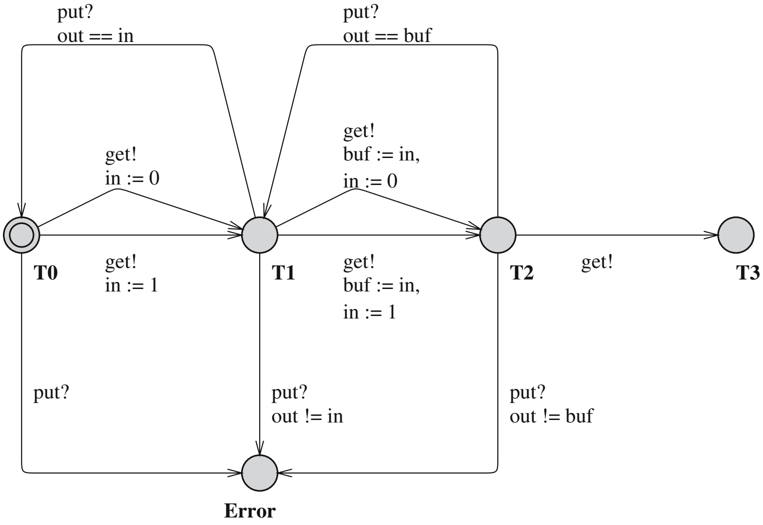

Fig. 10. Tester

bit that has been accepted by the **Coder** but not yet delivered by the **Decoder**. While searching for error scenarios that arise for parameter assignments that do not satisfy the constraints, we will encounter instances of our model in which two bits can be inside the protocol. We felt no need to model a tester that can handle situations in which three or more bits are sent but not yet received. Whenever the protocol (the **Decoder**) produces an output, the **Tester** checks whether this is the expected value. If it is correct, the **Tester** forgets the value, otherwise it jumps to a special **Error** location. If the protocol is correct then the **Error** location can not be reached.

3.10. Uppaal analysis results

The set of reachable symbolic states of our model is relatively small, and for all properties and parameter assignments that we tried, Uppaal managed to establish validity or produced a counterexample within a second (running Upppaal version 3.4.7 on a standard PC). Some basic well-formedness properties that we tested are that the system contains no deadlocks, the coder never starts another voltage transition (edge) while the **Wire** automaton is still in its unstable location, and that there are never more than two bits in transit in the protocol:

```
A[] not (deadlock or Wire.W2 or Tester.T3).
```

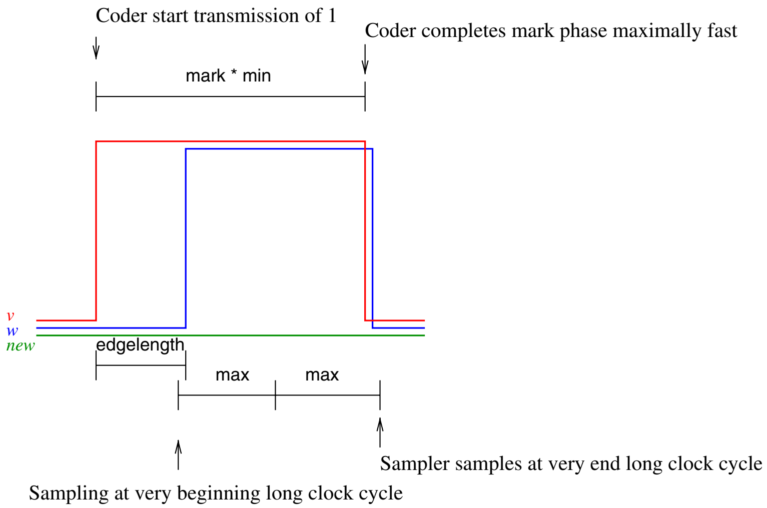
Coder start transmission of 1
Coder completes mark phase maximally fast
mark * min
v
w
new
edgelength
max
max
Sampler samples at very end long clock cycle
Sampling at very beginning long clock cycle

Fig. 11. First error scenario: decoder misses edge at beginning of cell

But the key correctness property, of course, is that the **Tester** never enters the **Error** location:

```
A[] not (Tester.Error).
```

Whether these properties hold depends on the specific choice of the parameter values. Through playing with different parameter assignments, and replaying the error traces in the simulator, we discovered that there appear to be essentially three different scenarios that may lead the **Tester** to the **Error** location: (1) the decoder may miss the edge at the beginning of a cell, (2) the decoder may sample too early, or (3) it may sample too late.

The first error scenario is illustrated in Figure 11. In this scenario, the coder transmits a 1 and passes through the mark phase (location **C2**) maximally fast. This means that $\mathsf{mark} \cdot \mathsf{min}$ time units after the *edge* event that marks the beginning of the cell we already see the *edge* event that marks the end of the mark phase (the line labeled v in Fig. 11). We assume that after the first edge the wire remains unchanged maximally long, that is $\mathsf{edgelength}$ time units, whereas after the second edge the wire immediately takes the new value (the line labeled w in Fig. 11). Now the decoder may altogether miss the voltage change on the wire if it (1) operates maximally slow for two clock cycles, (2) samples at the very beginning of the first clock cycle, just before the value of w changes, (3) samples again at the very end of the second cycle, right after the value of w has changed again. In order to avoid this error scenario, the following constraint on the parameters must be met, which ensures that an edge at the beginning of a cell will always be detected by the decoder:

$$\mathsf{mark} \cdot \mathsf{min} > 2 \cdot \mathsf{max} + \mathsf{edgelength} \qquad (1)$$

The second error scenario, in which the decoder samples too early, is illustrated in Fig. 12. In this scenario, the **Coder** operates maximally slow whereas the decoder operates maximally fast. The **Coder** transmits a 1 and remains in the mark phase maximally long, that is $\mathsf{mark} \cdot \mathsf{max}$ time units elapse between the two *edge* events (line labeled v in Fig. 12). This time, the wire immediately takes on the new value after the first edge, and sticks to the previous value maximally long after the second edge (line labeled w in Fig. 12). The decoder immediately detects the first edge and operates maximally fast. This means that the clock cycle in which it samples the value that will determine whether a 0 or a 1 will be decoded starts after $(\mathsf{sample} - 1) \cdot \mathsf{min}$ time units. If we assume that sampling takes place at the very beginning of this clock cycle, then the wrong bit (namely 0 instead of 1) will be decoded if the sampling takes place right before w changes its value for the second time. In order to avoid this error scenario, the following constraint on the parameters must be met:

$$(\mathsf{sample} - 1) \cdot \mathsf{min} > \mathsf{mark} \cdot \mathsf{max} + \mathsf{edgelength} \qquad (2)$$

This constraint ensures that the decoder will not sample too early.

The third error scenario, in which the decoder samples too late, is illustrated in Fig. 13. In this scenario the **Coder** operates maximally fast whereas the decoder is maximally slow: at the point where the decoder samples the

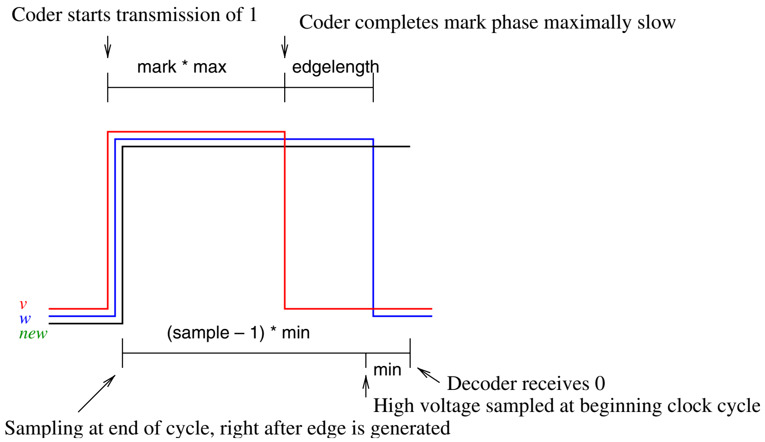
Coder starts transmission of 1
Coder completes mark phase maximally slow
mark * max
edgelength
v
w
new
(sample – 1) * min
min
Decoder receives 0
High voltage sampled at beginning clock cycle
Sampling at end of cycle, right after edge is generated

Fig. 12. Second error scenario: decoder samples too early

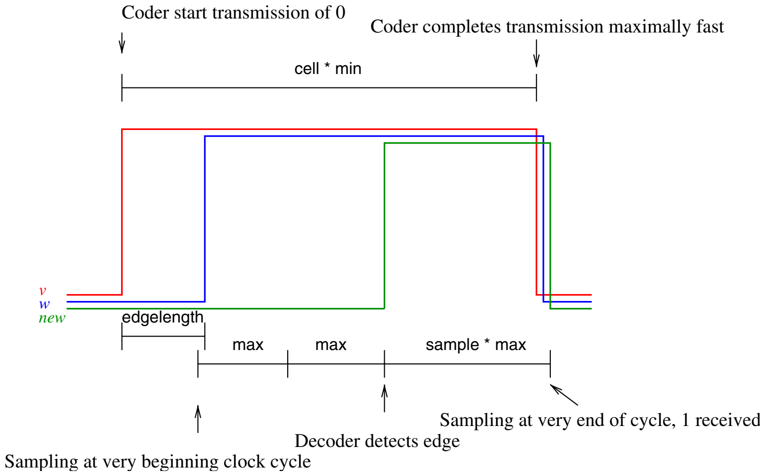
Coder start transmission of 0
Coder completes transmission maximally fast
cell * min
v
w
new
edgelength
max
max
sample * max
Sampling at very end of cycle, 1 received
Decoder detects edge
Sampling at very beginning clock cycle

Fig. 13. Third error scenario: decoder samples too late

coder has already started with the transmission of the next bit. In order to avoid this error scenario, the following constraint on the parameters must be met, which ensures that the decoder does not sample too late:

$$\mathsf{cell} \cdot \mathsf{min} > (\mathsf{sample} + 2) \cdot \mathsf{max} + \mathsf{edgelength} \tag{3}$$

An obvious question that arises is whether the three constraints introduced above are enough to ensure correctness: is the error location unreachable for all parameter assignments that satisfy constraints (1), (2) and (3)? This question can not be answered using UPPAAL, since UPPAAL can only compute the set of reachable states for a fixed parameter assignment, and there are infinitely many parameter assignments that satisfy the three constraints. Using deductive verification and the theorem prover PVS, we will establish in the next two sections that the three constraints together indeed guarantee correctness.

4. Translating the UPPAAL model into PVS

For the verification of the correctness of the biphase mark protocol with the given parameter constraints in a symbolic fashion we use the theorem prover PVS, which is a higher-order logic theorem prover developed by SRI [ORSH95]. We employ a framework in PVS that provides us with the standard definitions for automata and a guideline as to how to translate automata from UPPAAL to PVS. The translation is intended to ensure that the

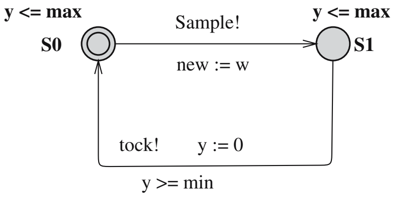

Fig. 14. Product of automata for sampler and decoder clock

PVS model closely resembles the UPPAAL model, so that it is easy to validate and to propagate changes from one model to the other.

An automaton in UPPAAL consists of a number of locations, some state variables and clocks, and transitions (labeled arrows) from one location to another; the transitions may also be labeled with assignments to the state variables and clocks, and they may be annotated with guards. Our translation deals with each of these parts of the automaton in turn, yielding a PVS model containing locations, state variables, and transitions. The translation of the automata in our UPPAAL model from diagrams into our PVS framework *in general* proceeds as follows:

1. The locations and state variables of the UPPAAL model are modeled in PVS as enumerations and records, respectively.
2. Anonymous transitions (transitions not labeled with an action label) need to be labeled to distinguish them. There are none in the UPPAAL model we have, since all of its transitions are labeled.
3. Our approach focuses on *local* translations – each automaton is translated into PVS with no regard for the context it will be placed in. Shared variables complicate this, because they require that two independently translation automata synchronize on an otherwise invisible action (i.e. assignments to the shared variables). Therefore we make that synchronization explicit. Shared variables are replaced by local variables whose values are synchronized via a parameter of the transition. For instance, the get! action is replaced by a get!(b) where b is the value generated by the tester and read by the coder.
4. The union of the sets of events of all the automata in the system is used as the global set of events. Explicit time delay is included as `delay(d)`.
5. For each automaton, we define a local state and a local transition relation, as well as the local initial state. The local transition relation must deal with the *global* set of events – this is because our translation is a simple one and does not take into account which events are relevant for which automata. No change should occur in response to events not used locally in the automaton.
6. The parallel composition of the automata is constructed by hand. The global state is a record containing the local states of each automaton. The global initial state is exactly the product of the local initial states.
7. The global transition relation applies each local transitions to the appropriate local state; since all local transitions are given the global event, synchronization on shared events in the run is obtained .

We will perform each of these steps for the model of the biphase mark protocol in the following sections. There is one place where we diverge from the UPPAAL model in Sect. 3: we translate the *product* automaton of the sampler and the second clock, instead of translating each individually. The reason for doing so is that the shared variable s, which ensures strict alternation of actions in the automata, is not used in a way that our shared-variable translation can deal with. It is, in the end, simpler to translate the product automaton than to invent a complicated translation that can deal with the shared variable.

4.1. Merging automata

The product automaton of the sampler and the second clock is easy to calculate, and yields an UPPAAL diagram like the one in Fig. 14. We use this automaton instead of the two separate ones because *both* automata write to the shared variable s, which makes our simple-and-straightforward translation to PVS inapplicable.

In this merged automaton, we have two locations, that represent the state $\mathsf{s} = 0$ and $\mathsf{s} = 1$ of the separate automata; again, Sample! and tock! events occur in turn, with no more than max time between Uppaal x_2 tock!s.

```
BMActions : DATATYPE BEGIN
    tock(b:bool)       : tock?    % Sampler clock tick
    get(b:bool)        : get?     % Get message bit
    put(b:bool)        : put?     % Put message bit
    sample(b:bool)     : sample?  % Sample from wire
    tick               : tick?    % Sender clock tick
    edge               : edge?    % Coder inverts wire value
    fuzz               : fuzz?    % Wire during edge
    settle             : settle?  % Wire stabilizes to value
    delay(d:posreal)   : delay?   % Time delay
END BMActions
```

Fig. 15. PVS Datatype for the Events of the biphase mark protocol

4.2. Removing shared variables

There are a few shared variables in the UPPAAL model, as shown in Fig. 2. These are:

- in, between coder and tester.
- w, between wire and sampler.
- new, between sampler and decoder.
- out, between decoder and tester.
- s, the variable shared between the sampler and the second clock, is *not* needed, since we use the product of those two automata instead.

Each of the uses, (reads or writes), of one of these shared variables can be changed into a parameterized event so that all variables become local. Consider in, shared between the coder and the tester. The tester nondeterministically sets the value of in on several transitions labeled with get!. These can be replaced by a parameterized get!(b) that represents the shared assignment in := b. In the coder, there is only one transition labeled get?, and the (shared) value of in is used in urgent transitions immediately afterwards. We replace it by get?(b), and save the value in a local variable in. It is straightforward to prove that the values of both local variables in in the two automata are always equal, which is what we would expect of a shared variable.

Similarly, the value w that is read from the wire by the sampler can be added as a parameter to the action Sample!. The wire's local transition allows a Sample! transition with parameter *b* only when the wire variable w has the value *b*. The sampler's transition tock! triggers a read of the value of new in the decoder. We can add the value of new as a parameter to it and use that value where needed.

Finally, the value of out is written by the decoder when it does a put! action, and the tester reads the value immediately. Here too, adding a parameter to pass the value put by the decoder removes the need for a shared variable.

4.3. Representing events

The collection of events in PVS is represented by an abstract datatype – this means that we automatically have axioms that the events are distinct, as one would expect from an enumeration of all the distinct kinds of events. The PVS code is shown in Fig. 15. We see the four parameterized actions, two actions without parameters, the two broadcast actions with no parameters (settle and fuzz), and the additional delay event representing the passage of time. Note that delay is parameterized by a posreal, i.e. a positive and non-zero amount of time. This means that in our PVS model, there are no events which delay by zero time. This is a subtle difference with the UPPAAL semantics, where zero delays are possible (they do not change the state or the value of any clocks though, so their deletion from the PVS model does no harm).

4.4. Local states and transitions

The aim of the translation into PVS is to prove that the parameter constraints that have been deduced in Sect. 3.10 are correct, i.e. that *all* choices of parameters within those constraints yield a correct protocol. We attempt to remain as close as possible to the UPPAAL model, so that it is intuitively clear that the PVS model represents the diagrams accurately. To this end we will do the following for each automaton: define an enumeration of the

```
ClockState : TYPE+ = [# x:Time #] ;

ClockTransition(s:ClockState,a:BMActions,s_:ClockState) : bool =
    CASES a OF
        delay(d) : s_‘x <= max AND s_ = s WITH [ x:= s‘x+d ],
        tick     : s‘x >= min AND s_ = s WITH [ x:=0 ]
    ELSE           s_ = s
    ENDCASES ;
```

Fig. 16. PVS code for the Clock’s state and transition relation

locations of the automaton (if it has more than one) so that we can refer to those locations by name; define a record that holds the location of the automaton and its local variables and clocks; define a transition relation on the local state where the primary selection is by event. This means that the transition relation is written in PVS like “if event is delay, then do this; else, if event is get, then do this ; . . . ” – the uniformity of expression defining the transition relation is important in partially automating proofs over the automaton.

The remainder of this section shows the translation of specific automata – the Clock, the Coder – in more detail.

The Clock: The clock has only one location, and a single clock named x, so the record structure for the state of the clock is very simple: a single field for the clock x, as can be seen in Fig. 16. One might argue that this representation could be simplified, down to identifying the state of the Clock with the value of the clock variable x, but we believe that consistency in structure is important for the automation of proofs.

There are two different transitions that the clock can take: time can pass (subject to the location invariant), or the clock can tick (subject to the transition guard). The PVS code for this does a case distinction on the event in order to determine whether a given state-transition is allowed. The PVS code for the transition is shown in Fig. 16. The PVS code defines a record type ClockState (the [# #] indicate a record) with a single field x for the clock. The transition relation for the clock is named ClockTransition and is a function of three variables, yielding a Boolean value. The parameters are s and s_, the local *from* state and the local *to* state, while a is the global event that occurs; the transition relation must state whether the transition from local state s to s_ when the system performs an action a is allowed. The transition relation is defined using a CASES statement, where we can list the events that change the local state of the clock. We see the use of the ELSE clause in the CASES expression in order to deal with “all-the-events-not-mentioned-yet.” The CASES statement is required to be total by PVS, so we use the ELSE clause to make it so. It is important to state that the state stays the same (s_ = s) when unhandled events occur, since otherwise the state *is* allowed to change nondeterministically when other automata perform an action.

The two events that are relevant for the Clock automaton are handled by writing the precondition (either the location invariant or the transition guard) in conjunction with an expression stating how the local variables should be updated. The update expression is typically written as s_ = s WITH [E]. The expression s WITH [E] has the value of the record s with the fields named in E updated by assignment; we read this as calculating the new state based on s and asserting that state s_ *is* that state. For the delay event, we need to check that the location invariant is not violated by a delay of d time, and the clock x must advance by d for the transition to be acceptable. In a similar vein, the transition for tick conjoins the transition guard with the resetting of clock x.

Coder: The structure of the coder is far complex than that of the Clock. There are only three event types (get, edge, tick) that need to be handled, but since the event labels occur multiple times in the diagram the expressions stating legality of a particular state-transition are more complex. The presence of urgent locations adds the delay event to the list-of-events-to-handle. In Fig. 17 we see that in has become iv – this is because in is a reserved word in PVS. The value iv is represented by a Boolean value; 0 or 1 would serve as well, but require an additional type definition. State component n is declared to be below[cell], which means $n <$ cell; PVS requires that we prove this (which is trivial and done automatically with typecheck-prove).

The transition relation for the Coder is fairly straightforward. Delay events cannot happen when the Coder is in locations c0, c1 or c4 since these are urgent locations (and recall that delays are non-zero in our PVS model), but the state of the coder must stay the same when delay actions do occur otherwise. The PVS code is shown in Fig. 18.

The event get? can only occur when the Coder is in location c0, and it is accompanied by a change in location and saving the parameter value to a local variable. This is straightforward enough. The edge! event can occur in

```
CoderLocations : TYPE+ = { c0, c1, c2, c3, c4 } ;
CoderState : TYPE+ = [#
    loc : CoderLocations,
    n   : below[cell],
    iv  : bool
    #] ;
```

Fig. 17. PVS code for the Coder's state

```
CoderTransition(s:CoderState,a:BMActions,s_:CoderState) : bool =
    CASES a OF
        delay(d) : ( s`loc = c2 OR s`loc = c3 ) AND s_ = s,
        get(b)   : s`loc = c0 AND s_ = s WITH [ loc:=c1, iv:=b ],
        edge     : ( s`loc = c1 AND s`iv AND
                     s_ = s WITH [ loc:=c2 ] ) OR
                   ( s`loc = c1 AND NOT s`iv AND
                     s_ = s WITH [ loc:=c3 ] ) OR
                   ( s`loc = c4 AND
                     s_ = s WITH [ loc:=c3 ] ) ,
        tick     : ( s`loc = c2 AND s`n < mark-1 AND
                     s_ = s WITH [ n:=s`n+1 ] ) OR
                   ( s`loc = c2 AND s`n = mark-1 AND
                     s_ = s WITH [ loc:=c4, n:=s`n+1 ] ) OR
                   ( s`loc = c3 AND s`n < cell-1 AND
                     s_ = s WITH [ n:=s`n+1 ] ) OR
                   ( s`loc = c3 AND s`n = cell-1 AND
                     s_ = s WITH [ loc:=c0, n:=0 ] )
    ELSE           s_ = s
    ENDCASES ;
```

Fig. 18. PVS code for the Coder's transition relation

one of three places; we write a disjunction of the transition expressions for each of those three distinct transitions. Each one of those transition expressions – as usual – checks that the initial location is correct, checks the guard on the transition, and states what the resulting state must be. Finally tick? occurs in four places in the diagram, and each of these is dealt with similarly.

Other automata: The other automata in the system (the wire, the sampler with the second clock, the decoder and the tester) are all straightforward to translate – the state consists of a few fields for the local variables, and the transitions are all of types similar to what we have already described. The complete PVS specification for the automata can be obtained from the URL mentioned in the introduction.

4.5. Global states and transitions

The global state of the entire system is of course the product of all the local states of the constituent automata. The easiest way to achieve this in PVS is to define a new record with one field for each of the constituent automata, as shown in Fig. 19. This global state contains *no* global variables, and hence the composition of the transition relations is straightfoward as well.

The transition relation for the global state is the conjunction of the local transitions for each automaton applied to the local states; the event a is passed to each local transition, along with the relevant local state. By having a structured and straightforward representation of the global state, the global transition becomes straightforward

```
GlobalState : TYPE+ = [#
    clock   : ClockState,
    coder   : CoderState,
    wire    : WireState,
    sampler : SamplerState,
    decoder : DecoderState,
    tester  : TesterState
    #] ;
```

Fig. 19. Global state of the system model of the biphase mark protocol

```
GlobalTransition(s:GlobalState,a:BMActions,s_:GlobalState) : bool =
    ClockTransition(s`clock,a,s_`clock) AND
    CoderTransition(s`coder,a,s_`coder) AND
    WireTransition(s`wire,a,s_`wire) AND
    SamplerTransition(s`sampler,a,s_`sampler) AND
    DecoderTransition(s`decoder,a,s_`decoder) AND
    TesterTransition(s`tester,a,s_`tester)
```

Fig. 20. Global transition relation of the biphase mark protocol

```
I : THEOREM
    LET T = R`states(i)`tester IN
    NOT (T`loc = t3 OR T`loc = error ) ;
```

Fig. 21. PVS code for main correctness theorem. The expressions for `T` represents the state of the Tester at index i of run R, while `T’loc` represents the location the Tester is in

as well and we can later apply some automation to the calculation of next-states in the executions of the global automaton. The global transition relation is shown in Fig. 20.

5. Proving correctness, with variations

The proof of the correctness of the biphase mark protocol was fairly straightforward, in large part thanks to the invariants turned up by the experimentation with UPPAAL. The additional verification found one omission in an invariant, which was repaired by strengthening it; this enabled us to prove the correctness of all instances of the protocol within the parameter space.

The proof in PVS of the correctness of the protocol and the necessity of the given bounds on the parameters is purely symbolic, and shows that *every* instance that falls within the parameter constraints suggested by the UPPAAL analysis is correct. The proof proceeds by collecting 37 invarants and verifying each invariant in turn; together, these invariants imply the global correctness of the protocol.

The global statement of correctness is much as in the UPPAAL verification: in all executions of the automata for the biphase mark protocol, there is no state of the system in which the tester automaton is in state t3, or in state error. In PVS this appears as the theorem shown in Fig. 21. In the theorem, the `LET` expression is substituted into the remainder of the theorem (after the `IN`), and the whole theorem is implicitly universally quantified over any unbound variables (in this case, index i and run R).

The UPPAAL verification – in particular the invariants checked there – gives a guideline for the formal verification of the protocol in PVS, but it is not a road map. Indeed, as far as UPPAAL is concerned, we need not bother with any invariant other than that t3 and error are unreachable. This is both the strength and the weakness of verification through UPPAAL – we verify a single instance, or several instances, and though the *truth* of UPPAAL's assertion that location t3 is unreachable does not change, we cannot see how this truth is arrived at, nor what the most general bounds of the parameters of the protocol might be.

Section 5.1 details the structure of the proof and the approach used, while Sect. 5.2 shows how some automation can be introduced into the proofs in order to shorten them and make them more understandable to the human reader. Section 5.3 examines the effect of introducing automation into the proofs.

5.1. Structure of the invariant proof

With the model as given the correctness condition is expressed in terms of the reachability of locations t3 and error. The correctness condition is given as an invariant on the state of the system, as already shown in Fig. 21. There are 37 invariants in all, each of which is proved by induction on the sequence of steps in a run of the automaton. The invariants are grouped as follows (Figs. 23, 24 on page 448 show more detail):

- Invariants of the clock or coder in isolation, named **P**x (6 invariants).
- Invariants of the wire, sampler, or decoder in isolation, named **Q**x (6).
- Invariants of pairs of automata, named **R**x (4).
- Invariants of the coder, clock and wire in parallel, named **T**x (5).
- Invariants of all but the tester in parallel, named **G**x (9).

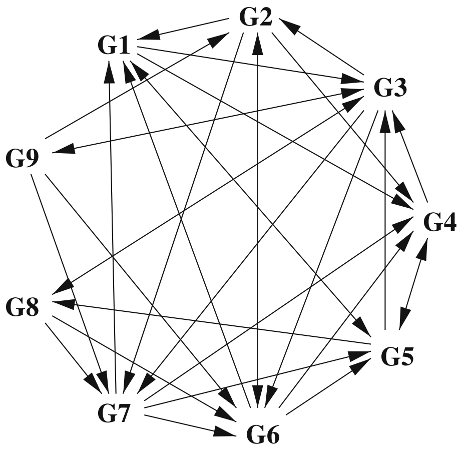

Fig. 22. The interdependence of lemmata in group **G**. An arrow from lemma $\mathbf{G}_i$ to $\mathbf{G}_j$ means that the proof of lemma i needs the result of lemma j

- Invariants of the system as a whole, named **H***x* (6).
- The global correctness invariant, **I** (1).

These 37 invariants were checked with UPPAAL on some instances of the protocol before beginning the PVS verification. In the PVS proof, some invariants were re-ordered (since the proof of **P5** needed the result of **P6**, for instance), and some corollaries were introduced to make proofs shorter. In the **P**, **Q** and **R** groups, each invariant was proved individually. Groups **T**, **G** and **H** were proved simultaneously with the following approach (which is tailored to being convenient in PVS – mathematically, we are merely showing that the group of invariants is inductive):

1. For each invariant I_k to be proven in the group, define a predicate $I_k(R, i)$ that takes an automaton run R and an index i as parameters that asserts the invariant property on the ith state of the run R.
2. For each invariant I_k, write and prove a lemma L_k for the induction step of I_k as follows (assume n invariants in the group):

$$I_1(R,i) \wedge I_2(R,i) \wedge \cdots \wedge I_n(R,i) \Rightarrow I_k(R, i+1)$$

3. Finally, define a lemma I for the group as a whole:

$$\forall i \;.\; I_1(R,i) \wedge I_2(R,i) \wedge \cdots \wedge I_n(R,i)$$

The proof of this lemma is simple: check the base case of the induction, (i.e. that in all initial states, each invariant holds individually), and for the induction step use lemmata L_1 through L_n.

This approach can be used to find the interdependency graph of the invariants as well, by determining which invariant-predicates are not used in each proof. Figure 22 shows how the invariants of group **G** are interdependent – this also makes clear that no smaller group could be constructed that is still provable.

The "simple" invariants in groups **P**, **Q** and **R** are the sixteen invariants shown in Fig. 23. The proof of each of these is fairly straightforward: induction on the index of the state the invariant is applied to. The base case is trivial and solved with `(grind)`, the induction step requires checking the relevant transitions, which means that the global transition relation `GlobalEffect` needs to be introduced and expanded to the local transitions. Since these steps are the same for every proof, we define a simple strategy (scheme for automation) called `(auto-start)` that starts off an inductive proof by dealing with the base cases and the introduction of the local transitions (see Sect. 5.2 for more details on the additional automation).

These 16 invariants, along with one additional support lemma, take 204 proof steps to prove. Each invariant depends only on the invariants preceding it in the list (hence **P6** and **P5** are reversed, since the naming of the invariants was established before the proofs were begun). Additional automation can reduce the number of steps taken considerably (i.e. by half), which we will examine shortly.

The remaining invariants are divided into four groups. The groups **T**, **G** and **H** are intra-dependent, which was partly illustrated in Fig. 22, while group **I** contains only a single invariant. Figure 24 shows the invariants.

	Clock and Coder
P1	$0 \leq x \leq \mathsf{max}$
P2	$\mathsf{c}_0 \vee \mathsf{c}_1 \Rightarrow n = 0$
P3	$\mathsf{c}_2 \Rightarrow in \wedge 0 \leq n < \mathsf{mark}$
P4	$\mathsf{c}_3 \Rightarrow 0 \leq n < \mathsf{cell}$
P6	$\mathsf{c}_4 \Rightarrow in \wedge n = \mathsf{mark}$
P5	$\mathsf{c}_3 \wedge n < \mathsf{mark} \Rightarrow \neg in$
	Wire, Sampler and Decoder
Q1	$0 \leq z$
Q2	$\mathsf{w}_1 \Rightarrow z \leq \mathsf{edgelength}$
Q3	$0 \leq y \leq \mathsf{max}$
Q4	$\mathsf{d}_0 \Rightarrow m = 0$
Q5	$\mathsf{d}_1 \Rightarrow 0 \leq m < \mathsf{sample}$
Q6	$\mathsf{d}_2 \Rightarrow m = \mathsf{sample}$
	Automaton Pairs
R1	$\mathsf{c}_0 \vee \mathsf{c}_1 \vee \mathsf{c}_4 \Rightarrow x = 0$
R2	$\mathsf{s}_0 \wedge z > y + \mathsf{edgelength} + \mathsf{max} \Rightarrow w = new$
R2	$\mathsf{s}_1 \wedge z > y + \mathsf{edgelength} \Rightarrow w = new$
R3	$\mathsf{d}_2 \Rightarrow y = 0$
R4	$\mathsf{w}_0 \Rightarrow v = w$

Fig. 23. The "simple" invariants of at most two automata

The effort for each intra-dependent group is far greater than the effort for the simpler invariants listed above. The **T** group of five invariants requires 432 proof steps – again, with additional automation this can be reduced considerably.

The **G** group of invariants is by far the most complicated of the nests of interdependent invariants, and while proving it the original invariants suggested by the UPPAAL tests were found to be insufficiently strong to prove the entire nest. The invariant **G1** needs the additional condition

$$\mathsf{mark} \leqslant n \Rightarrow \mathsf{sample} \cdot \mathsf{min} - \mathsf{mark} \cdot \mathsf{max} \leqslant z$$

This condition was discovered after staring at the proofs – all of which could not be finished because of the lack of this information – for a few days. Once that was done, the proofs were fairly straightforward again, with only the question of which invariants were dependent on which others.

Although invariants **H4**, **H5** and **H6** can be proved easily from **H1**–**H3** and the **G** invariants, it turned out that **H1**–**H3** need the later **H** invariants in their proofs, which made this a new (though simple) nest of interdependent invariants.

Finally, **I** is an almost trivial conclusion of **H1**–**H3** and **H6**.

Initial proof statistics, of the hand-made proof with little automation, are shown in Fig. 25 (these are entirely dependent on the PVS user that creates them, though). The statistics show that all the proofs together use only 28 different proof commands, one of which is a locally defined strategy for the purpose of initiating automaton invariant proofs `(auto-start)`. Four are sequent management commands not immediately relevant for the proofs themselves (these are `(name)`,`(hide)`, `(reveal)` and `(delete)`). Three are proof commands automatically used by PVS for proving TCCs, (Type-Check Conditions, normally inserted by PVS when it cannot automatically deduce the type of an expression). One proof command is inserted by PVS automatically to finish a trivial proof (one of the form $P \Rightarrow P$). This leaves 19 different commands that are actually typed by the user; the vast majority are `(assert)` and `(flatten)`, which follow from the habits of the PVS user who made these particular proofs.

It should be clear that the amount of effort to do the proofs is enormous compared to the effort involved in the UPPAAL verification. The main reason for this is that initially it is not clear how each proof should proceed, and there is little support built in to PVS for the kind of proofs that need to be done. Eventually, we see that the structures of the proofs are remarkably regular, and can develop some reusable automation for dealing with them.

There are several existing implementations of additional automation for proofs in a specific framework, both in PVS and outside of PVS. One example of far-ranging automation is ACL2, which is a highly automated

	Coder, Clock and Wire
T5	$\neg \mathsf{w}_2$
T6	$(\mathsf{c}_0 \lor \mathsf{c}_1 \lor \mathsf{c}_4) \Rightarrow \mathsf{w}_0$
T7	$\mathsf{c}_2 \lor (\mathsf{c}_3 \land in) \Rightarrow n \cdot \mathsf{min} \leq z - x \leq n \cdot \mathsf{max}$
T8	$\mathsf{c}_4 \Rightarrow \mathsf{mark} \cdot \mathsf{min} \leq z \leq \mathsf{mark} \cdot \mathsf{max}$
T9	$\mathsf{c}_3 \land in \land \mathsf{mark} \leq n \Rightarrow$ $(n - \mathsf{mark}) \cdot \mathsf{min} \leq z - x \leq (n - \mathsf{mark}) \cdot \mathsf{max}$
	Global (except Tester)
G1	$\mathsf{d}_0 \Rightarrow \neg \mathsf{c}_4 \land (v = new \lor new = old)$
G2	$\mathsf{d}_1 \Rightarrow \mathsf{c}_2 \lor \mathsf{c}_3 \lor \mathsf{c}_4$
G3	$\mathsf{d}_2 \Rightarrow \mathsf{c}_3 \land in = out \land v = new \land new = old$
G4	$\mathsf{d}_0 \land (\mathsf{c}_0 \lor \mathsf{c}_1 \lor (\mathsf{c}_3 \land \mathsf{mark} \leq n)) \Rightarrow v = old$
G5	$\mathsf{d}_0 \land (\mathsf{c}_2 \lor \mathsf{c}_3) \land n < \mathsf{mark} \Rightarrow v \neq old \land z \leq y + \mathsf{max} + \mathsf{edgelength}$
G6	$\mathsf{d}_1 \land (\neg in \lor \mathsf{c}_2 \lor \mathsf{c}_4) \Rightarrow$ $v = old \land (m \cdot \mathsf{min} \leq z - y \leq (m+2) \cdot \mathsf{max} + \mathsf{edgelength}$
G7	$\mathsf{d}_1 \land \mathsf{c}_3 \land in \Rightarrow v \neq old \land$ $m \cdot \mathsf{min} - \mathsf{mark} \cdot \mathsf{max} \leq z - y$ $\leq (m+2) \cdot \mathsf{max} - \mathsf{mark} \cdot \mathsf{min} + \mathsf{edgelength}$
G8	$\mathsf{d}_2 \land \neg in \Rightarrow \mathsf{sample} \cdot \mathsf{min} \leq z < \mathsf{cell} \cdot \mathsf{min}$
G9	$\mathsf{d}_2 \land in \Rightarrow \mathsf{sample} \cdot \mathsf{min} - \mathsf{mark} \cdot \mathsf{max} \leq z < (\mathsf{cell} - \mathsf{mark}) \cdot \mathsf{min}$
	Global
H1	$\mathsf{d}_1 \lor \mathsf{d}_2 \Rightarrow \mathsf{t}_1$
H2	$\mathsf{d}_0 \land \mathsf{mark} \leq n \Rightarrow \mathsf{t}_0$
H3	$\mathsf{d}_0 \land \mathsf{mark} > n \Rightarrow \mathsf{c}_0 \lor \mathsf{t}_1$
H4	$\mathsf{c}_3 \Rightarrow \mathsf{t}_0 \lor \mathsf{t}_1$
H5	$\mathsf{c}_1 \lor \mathsf{c}_2 \lor \mathsf{c}_4 \Rightarrow \mathsf{t}_1$
H6	$\mathsf{c}_0 \Rightarrow \mathsf{t}_0$
	Global Correctness
I	$\neg(\mathsf{t}_2 \lor \mathsf{t}_3 \lor \mathsf{error})$

Fig. 24. Invariants for three or more automata in parallel

first order theorem prover which has been used for the mechanical verification of microprocessors [SH98]. Similar (but much smaller-scale) microprocessor verification has been done in PVS, although with no automation [Cyr93]. Protocol verification using roughly the same framework as we have used here can be found in [CHdV03]. Automation of PVS proofs in a specific context is mentioned briefly in [Gri00], Sect. 6.6.1; it is a shame that such improvements have not percolated into the standard PVS distribution. The TAME modeling environment [AHS98] offers automation for timed automata models in PVS.

5.2. Introducing automation to the proofs

With the statistics of Fig. 25 as a baseline, we can attempt to slim down the proofs by introducing additional automation. Two examples that occur quite frequently in the proofs are:

- **Using `(case)` to split up an implication.** The PVS command `(split)` splits a sequent with an implication as follows:

  $(A \Rightarrow B) \vdash C \quad \text{becomes} \quad B \vdash C \land \vdash A$

  This throws away the fact that A holds in the left branch of the proof; hence, we often use `(case)` and `(assert)` in order to achieve:

  $(A \Rightarrow B) \vdash C \quad \text{becomes} \quad A, B \vdash C \land \vdash A$

  By creating a tiny strategy `(split*)` that does this automatically, we achieve two things:

  1. The proof becomes shorter (in terms of user-entered steps)

PVS Command	Count	Note
EXISTENCE-TCC	1	Inserted by PVS
BETA	1	Reduce function application
INDUCT	2	For invariant `I`
SKOLEM	2	Manual instantiation
NAME	2	Introducing abbreviations
TYPEPRED	3	Needed for min < max
INST?	3	Automatic instantiation
ASSUMING-TCC	3	Inserted by PVS
IFF	5	Split boolean equivalence.
SUBTYPE-TCC	9	Inserted by PVS
AUTO-START	18	Partially automated strategy
GRIND	23	Prove automatically
HIDE	25	Sequent management
COMMENT	41	Sequent management
SKOLEM!	45	Automatic instantiation
REVEAL	49	Sequent management
LEMMA	62	Using invariants
INST	67	Manual instantiation
CASE	93	Introducing facts
GROUND	115	Decision procedures
PROPAX	124	Trivial decision procedures
REPLACE	131	Rewriting
LIFT-IF	139	Changing `CASES` to `IF`
DELETE	149	Sequent management
SPLIT	236	Changing `IF` to a split case
USE	265	Using invariants
EXPAND	728	Using definition
FLATTEN	748	Simplify sequent
ASSERT	1224	Simplify sequent
Total:	4313	

Fig. 25. Proof statistics for first attempt at invariants

2. It becomes clearer where this technique is used, i.e. it distinguishes these frivolous uses of `(case)` from ones where real new facts are introduced.

- **Splitting a `CASES` statement.** When confronted by a large `CASES` statement in a sequent (which is common in the proof of the biphase mark protocol, since we have many automata with fairly large transition relation expressions), it is often desirable to split it into one sequent for every case in the expression. This has the effect of examining each transition individually. Typically, the sequent appears as:

```
{-1} CASES R!1‘events(i!1) OF
       delay(d) : P1
       tick     : P2
     ...
```

This can be reduced to a collection of sequents, one for each case in the expression, each with a specific event and transition predicate. For instance, the second sequent to prove here would be

```
{-1} tick?(R!1‘events(i!1))
{-2} P2
     ...
```

This can be achieved by using `(lift-if)` to change the `CASES` statement into a collection of nested `IF`s, and then using `(split)` to split the `IF` statement repeatedly, with the liberal application of `(flatten)` and `(assert)` to massage the sequent into basic shape (or prove particular subgoals automatically). Automating this process in a single prover command `(auto-step)` gives us:

1. Shorter proofs
2. A more uniform structure of the proofs

Command	Before	After
EXISTENCE-TCC	1	1
INDUCT	1	1
SKOLEM	1	1
SPLIT*		1
ASSUMING-TCC	2	1
TYPEPRED	3	1
COMMENT		2
GRIND	9	18
SUBTYPE-TCC	9	9
CASE	10	6
PROPAX	12	2
HIDE	13	0
SPLIT	14	9
AUTO-START	16	22
SKOLEM!	17	11
LEMMA	18	9
INST	23	14
DELETE	34	17
AUTO-STEP		35
LIFT-IF	37	1
REVEAL	38	0
REPLACE	45	18
GROUND	59	23
USE	61	55
FLATTEN	75	32
EXPAND	111	24
ASSERT	217	77
Total:	826	390

Fig. 26. Proof statistics for groups **P**, **Q**, **R** and **T**, before and after automation

We gain additional flexibility by automatically expanding local transition statements and by using the names of local transitions, instead of formula numbers. A typical application of the resulting strategy is to expand and simplify the local transition for the Coder with `(auto-step ('c "Coder"))`.

After re-doing the proofs with the additional automation (and with the knowledge that the first run of a proof in PVS is nearly always a bit messy), the proof statistics become much smaller. For invariant groups **P**, **Q**, **R** and **T** results are shown in Fig. 26.

This 79% reduction in the number of proof steps is partly attributable to the increased automation afforded by the `(auto-step)` command. Some of the reduction can be attributed to the difference between *finding* a proof (the initial proof attempt) and *polishing* a proof for presentation. While applying the increased automation to the proof we also have the benefit of knowing how the proof is supposed to go, and we can judiciously prune the proof of less-than-optimal proof explorations. Additionally, it occurs fairly regularly that it is unclear whether some subtree in a proof can be proved easily; once the proof is done it is clear that `(grind)` would have done the job as well, so the re-run of a proof replaces whole subtrees with `(grind)`.

5.3. Summary of automation

Once the structure of the PVS proofs for the biphase mark protocol was clear, additional automation was introduced in order to reduce the amount of steps used in doing the proofs. The comparisons of numbers of proof steps with and without the automation made in the previous section suggest what *could have been*. Future proofs with a simular structure may also benefit from this automation, using the new proof commands that were introduced:

- `(split*)` Handle implications in a nicer way than `(split)`.
- `(auto-start)` Start an automaton proof by introducing local transition relations.
- `(auto-step)` Expand and rewrite a local transition relation.

There is a trade-off, though, when doing automated proofs, between brevity and comprehensibility. PVS's tremendously powerful `(grind)` command can reduce many proofs to one step, *once the proof has been found.*

Using `(grind)` when it is unclear that the lemma is sound is unwise, since it takes some time to grind its way through the proof, and then it can:

- Fail, returning you to the original proof state and requiring you to do the proof by hand anyway, or
- Give you 64 (or some other large number) bizarrely formed subgoals to prove.

Neither of these results of `(grind)` are really useful for advancing the proof itself. Therefore we feel that the use of `(grind)` should be restricted to those proofs that really are trivial. Somewhere in the middle lies the ideal, of a proof that is short enough to understand and not so thoroughly automated that it is unbelievable. The use of the Uppaal model gives similar results: we know something is true, but not necessarily *why* or whether the fact is interesting.

Consider invariant **G9**:

```
G9(R,i) : bool =
    LET D = R`states(i)`decoder, C = R`states(i)`coder,
        W = R`states(i)`wire, S = R`states(i)`sampler IN
    D`loc = d2 AND C`iv =>
    sample*min - mark*max <= W`z AND W`z < (cell-mark)*min ;
```

The important step in the proof of **G9** is the induction step, which is proven in the PVS lemma `Gi`:

```
Gi : LEMMA
    G9(R,i) AND G3(R,i) AND G7(R,i) AND G2(R,i) AND G6(R,i) => G9(R,i+1)
```

The proof itself uses the earlier invariant **Q3**, the parameter assumption from Eq. 3 (on page 441) and an additional lemma, called **Gi1**. The proof itself has the following structure:

- Start with `(auto-start)`.
- Expand **G9** itself.
- For each automaton, use `(auto-step)` to rewrite its local transitions and prove trivial subcases.
- Do a little rewriting and formula manipulation, introduce the additional invariants **Q3** and **Gi1** that are needed to prove each step.

Without automation, the proof of **G9** took 179 steps, exploring blind alleys, over-using `(assert)`, and doing formula manipulation the tedious way. With a little automation such as described in the previous section, the number of proof steps declined to 59. In this simplified proof the structure of examining each automaton's local transitions was very visible. Further reflection, though, shows that the proof can be reduced to 5 steps:

```
(AUTO-START T)                    % Deal with base case.
(LEMMA "Gi1" ("i" "j!1" "R" "R")) % Needed much later with this
(LEMMA "Q3" ("i" "j!1" "R" "R"))  % particular instantiation.
(USE "SampleEarlyEnough")
(GRIND)                           % Let PVS do the work.
```

This particular proof is a good example of a type of proof commonly found in mathematics texts: "Use lemmata `Gi1`, `Q3` and the parameter inequalities; the details are left to the interested reader." In our context the interested reader is the PVS theorem prover, which works out the details. Now, this proof might be good for verification purposes but it is certainly not the kind of proof one can write on first setting out to prove a property like **G9**. It is also not the kind of proof you would want to present as a didactic example to show the kind of reasoning needed in a particular domain, but again, as a succinct demonstration of truth it is fine.

This suggests that we can distinguish three flavors of proof, created while reasoning about the biphase mark protocol:

1. Exploratory proofs when we do not know how to prove the lemma – or even if the lemma is true. These proofs use mostly basic commands from the PVS proof language and are rather lengthy, although each step is very basic.
2. Polished proofs, using some automation that is built around the specific domain being studied and the framework that is in use. With suitable (not overly case-specific) automation, the size of proofs can be reduced over 50%, while their comprehensibility is improved because we can (for instance) replace a scattered collection of `(expand)`, `(lift-if)` and `(split)` with a single `(consider-each-local-transition)` proof command (although it is called `(auto-step)` in our automation attempt).

3. Proofs that are as short as possible, for the purpose of machine verification of the lemma. These are useful for re-checking a theory after changes have been incorporated, or as a basis for "details are left to the reader" expositions.

6. Playing with the parameter inequalities

Now that we have formally derived a number of constraints on the protocol parameters, it is interesting to explore the consequences of these results. An implementor of BMP will probably have limited influence on the values of min, max and $\mathsf{edgelength}$, but (s)he may freely choose the values of cell, mark and sample. For which values of these parameters is the bit rate maximal? Are the conventional implementation choices indeed the optimal ones?

Rather than the specific values for the lower bound min and upper bound max on the time between clock ticks, we find it convenient to consider the ratio

$$\rho = \frac{\mathsf{min}}{\mathsf{max}}.$$

Since $0 < \mathsf{min} \leqslant \mathsf{max}$, ratio ρ is contained in the interval (0, 1]. If $\rho = 1$ we have perfect hardware clocks, and the closer ρ gets to 0, the more unreliable the clocks are. We also normalize the time $\mathsf{edgelength}$ with respect to the maximum time between clock ticks:

$$E = \frac{\mathsf{edgelength}}{\mathsf{max}}.$$

So E specifies the number of clock cycles the signal may remain distorted after occurrence of an edge. Now we can rewrite the parameter constraints (1), (2) and (3) into:

$$\mathsf{mark} \cdot \rho > 2 + E \tag{4}$$
$$(\mathsf{sample} - 1) \cdot \rho > \mathsf{mark} + E \tag{5}$$
$$\mathsf{cell} \cdot \rho > \mathsf{sample} + 2 + E \tag{6}$$

Since $\rho \in (0, 1]$ and $E \geqslant 0$, inequality (4) implies $\mathsf{mark} > 2$. Using this fact in combination with inequality (5) implies $\mathsf{sample} > 3$. Substituting this in inequality (6) gives $\mathsf{cell} > 5$.

6.1. Minimizing the Cell Size (Assuming $E = 1$)

Moore [Moo94] assumed that the uncertain values read from the signal due to the presence of an edge are limited to the time-span of the cycle during which the edge was written, that is he assumed $\mathsf{edgelength} = \mathsf{max}$ or equivalently $E = 1$. With this additional assumption, the parameter inequalities further simplify to

$$\mathsf{mark} \cdot \rho > 3 \tag{7}$$
$$(\mathsf{sample} - 1) \cdot \rho > \mathsf{mark} + 1 \tag{8}$$
$$\mathsf{cell} \cdot \rho > \mathsf{sample} + 3 \tag{9}$$

Hence with ρ close to 1 the minimal values for the other parameters are

$$\mathsf{mark} = 4 \quad \mathsf{sample} = 7 \quad \mathsf{cell} = 11.$$

Implementors prefer to use instances of BMP with

$$\mathsf{cell} = 2 \cdot \mathsf{mark} \tag{10}$$

since this implies that the signal on the wire will be high approximately 50% of the time and low 50% of the time, which is desirable from an electrical engineering perspective ("DC balanced"). With this additional requirement, the minimal values become

$$\mathsf{mark} = 7 \quad \mathsf{sample} = 10 \quad \mathsf{cell} = 14.$$

These unconventional choices permit a faster bit rate (since fewer cycles are spent on each bit) than the conventional choices

$$\mathsf{mark} = 16 \quad \mathsf{sample} = 23 \quad \mathsf{cell} = 32, \text{ and}$$

cell	16	32	18	11	14
mark	8	16	5	4	7
sample	11	23	10	7	10
ρ	0.91	0.82	0.73	0.91	0.93

Fig. 27. Lower bound on ρ for some example configurations (with $E = 1$)

cell	16	32	18	11	14
mark	8	16	5	4	7
sample	11	23	10	7	10
E	1.989	5.977	2.994	1.988	1.985

Fig. 28. Upper bound on E for some example configurations (with $\rho = 0.999$)

$$\mathsf{mark} = 8 \quad \mathsf{sample} = 11 \quad \mathsf{cell} = 16.$$

The next lemma states that if we assume that the cell size is twice the mark size, inequality 4 becomes redundant.

Lemma 1 *Inequality* (4) *follows from* (*in*)*equalities* (10), (5), (6), $E \geqslant 0$ *and* $\rho \in (0, 1]$.

Proof. We derive:

$$\begin{aligned}
\mathsf{mark} \cdot \rho &\overset{(10),(6)}{>} \mathsf{sample} - \mathsf{mark} \cdot \rho + 2 + E \\
&\overset{\rho\in(0,1]}{\geqslant} \mathsf{sample} \cdot \rho - \rho - \mathsf{mark} + 2 + E \\
&\overset{(5)}{>} \mathsf{mark} + E - \mathsf{mark} + 2 + E \\
&\overset{E\geqslant 0}{\geqslant} 2 + E.
\end{aligned}$$

□

6.2. Maximizing the clock tolerance

By combining constraints (4), (5) and (6) we infer a lower bound on ρ, that is, the maximal tolerance on timing:

$$\rho > \max\left(\frac{2 + E}{\mathsf{mark}}, \frac{\mathsf{mark} + E}{\mathsf{sample} - 1}, \frac{\mathsf{sample} + 2 + E}{\mathsf{cell}}\right) \tag{11}$$

Figure 27 lists lower bounds for ρ for some example configurations, assuming $E = 1$. These numbers can be easily validated using the Uppaal model checker. Our results significantly improve on those of Moore [Moo94], who obtained (for a model that is less general) a lower bound of 0.95 for ρ for the 18-cycle version of BMP, and a lower bound of 0.97 for the conventional 32-cycle version. Typical clocks used in hardware are incorrect by less than 6×10^{-6} s per second [Cri89]. Thus,

$$\rho \geqslant \frac{1 - 6 \times 10^{-6}}{1 + 6 \times 10^{-6}} \approx 0.99999.$$

This means that in practice there is no need to optimize on the lower bound for ρ.

6.3. Maximizing the edge distortion tolerance

From a practical perspective, it is interesting to look for the maximal value for E, since, as we will see in the next subsection, this will allow us to optimize the bit rate. Using inequalities (4), (5) and (6) we infer that E may take any value as long as:

$$E < \min(\mathsf{mark} \cdot \rho - 2, (\mathsf{sample} - 1) \cdot \rho - \mathsf{mark}, \mathsf{cell} \cdot \rho - \mathsf{sample} - 2) \qquad (12)$$

Figure 28 lists upper bounds for E for some example configurations, taking a value 0.999 for ρ. If $\mathsf{cell} = 2 \cdot \mathsf{max}$ then, by Lemma 1, the minimal value for the right hand side of inequality (12) is reached by either the second or third subterm of the min-expression. Since sample occurs positively in the second term and negatively in the third term, the choice of a real number value for sample that maximizes E for this case is the one for which the second and third term are equal:

$$(\mathsf{sample} - 1) \cdot \rho - \mathsf{mark} = \mathsf{cell} \cdot \rho - \mathsf{sample} - 2.$$

The optimal (in the sense that it maximizes E) choice for sample therefore is

$$\frac{\mathsf{cell} \cdot \rho + \mathsf{mark} + \rho - 2}{1 + \rho}$$

This optimal value is typically slightly less than $\frac{3\mathsf{mark}-1}{2}$ since

$$\frac{3\mathsf{mark} - 1}{2} - \frac{\mathsf{cell} \cdot \rho + \mathsf{mark} + \rho - 2}{1 + \rho} = \frac{(1 - \rho)(m + 3)}{2(1 + \rho)} \geqslant 0$$

Using this observation, we may infer that (for realistic values of ρ and m, say $\rho \geqslant 0.999$ and $m \leqslant 1000$):

- If mark is odd then a strict upper bound on the value for E is $\frac{4\rho\mathsf{mark}-3\mathsf{mark}-3}{2}$. If we choose for sample the (integer) value $\frac{3\mathsf{mark}-1}{2}$ then E may take any value below this upper bound.
- If mark is even then a strict upper bound on the value for E is $\frac{3\rho\mathsf{mark}-2\mathsf{mark}-4\rho}{2}$. If we choose for sample the (integer) value $\frac{3\mathsf{mark}-2}{2}$ then E may take any value below this upper bound.

We write $E_{opt}(\mathsf{mark})$ for the upper bound on the value of E for mark size mark, and $\mathsf{sample}_{opt}(\mathsf{mark})$ for the optimal choice for sample given mark. In all examples of Fig. 28 with $\mathsf{cell} = 2 \cdot \mathsf{max}$ the actual value of sample equals the optimal value (in the sense that it maximizes the upper bound on E) that we derived.

6.4. Optimizing the bit rate

We can now generalize the results from Section 6.1 to a setting with arbitrary E. If we know E and ρ then in order to optimize the bit rate we need to find the instance of BMP with the smallest cell size that is correct. To obtain this instance, we just take the smallest m with $E_{\mathrm{opt}}(m) > E$ and then set cell to $2m$, mark to m, and sample to $\mathsf{sample}_{\mathrm{opt}}(m)$.

Based on our model we conclude that the 14-cycle instance of BMP is preferable over the 16-cycle instance that has been implemented in the Intel 82530 Serial Communications Controller. The 14-cycle version of the protocol allows for a more than 14% faster bit rate, but has basically the same tolerance for signal distortion following an edge. Also, the 30-cycle instance of BMP probably is preferable over the conventional 32-cycle version: it has almost the same tolerance for signal distortion after an edge ($E = 5.97$ if $\rho = 0.999$) but allows for a more than 6% faster bit rate. Note however that our model is quite abstract and ignores various engineering realities like metastability, reflection, noise and distortion. Like Moore [Moo94], we offer our model primarily as a catalyst for thought. It is up to the engineers to decide whether our model is accurate enough for the purposes at hand.

7. Related work and concluding remarks

Related Work A main source of inspiration for this article has been the work of Moore [Moo94]. The basic modelling assumptions that we use are similar to the ones proposed by Moore, although our model is somewhat more general: unlike Moore we allow for clock jitter in our model, and we also drop Moore's assumption that 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. Moore [Moo94] developed a general model of asynchronous communication and used this model to verify the correctness of 18 and 32 cycle instances of BMP. Interestingly, Moore did not succeed in establishing correctness of the 16 cycle instance that has been implemented by Intel. As we pointed out in Sect. 6.2, the bounds on timing uncertainty found by Moore are suboptimal.

Model checkers for timed and hybrid automata have been used successfully to analyze various physical level communication protocols for consumer electronics devices [BPV94, DY95, HWT95, BGK$^+$96, HSLL97]. Since these protocols typically use variations of biphase mark (e.g. Manchester) an obvious idea was to try to recast Moore's work in a setting of timed or hybrid automata. A first attempt in this direction was made by Ivanov and Griffioen [IG99], who automatically verified a few instances of BMP using the model checker HyTech. Their model is somewhat restrictive, however, since for instance sampling was only allowed at the end of a read cycle. In September 1999, the first author (FV) constructed the UPPAAL model that has been described in this paper (with only a few minor differences), and gave a presentation of this model and the derived parameter constraints during a symposium on the occasion of the retirement of Hans Peek as professor at the University of Nijmegen. After model and slides were made available on the web, several researchers took up the challenge to synthesize the parameter constraints automatically. Bensalem et al. [BBF$^+$00] propose algorithms and methods to compute invariants of infinite-state systems. Using their approach they managed to synthesize versions of the last two parameter constraints for a simplified version of our model in which the **Wire** and **Sampler** automaton have been left out. The first parameter constraint is not needed in the simplified model. Henzinger et al. [HPWT01] succeeded to partially synthesize our parameter constraints for BMP (they always had to fix some parameter) by running HyTech on a manually constructed abstraction of the model.

An independent line of research was carried out by Van Hung [HI96, Hun98]. In this work, the BMP has been modelled using Duration Calculus, and a full parameter analysis has been carried out with PVS. Van Hung models beginning and end of transmission, but assumes fixed clock rates (no jitter). The parameter inequalities discovered by Van Hung are similar to ours but with some "off by one" differences. Apparently, these differences are caused by some counterintuitive property of the Duration Calculus model: if the coder generates an edge, then the signal on the wire will be unreliable for E cycles (RR in Van Hung's terminology) starting from the last tick of the *receiver* clock. We believe our timed automaton model is more realistic.

Conclusions A fascinating question for us is whether our results about possible improvements of bit rates of the biphase mark protocol carry over from our model to the real world. Although we believe that our model accurately reflects the operation of some implementations of BPM (such as Intel 82530), there are other implementations in which the receiver operates in a slightly different manner. In the popular AMD 85C30 Serial Communications Controller [Adv92], for instance, clock information is recovered from the BPM signal using a digital phase-locked loop (DPLL). The DPLL is driven by a clock that is nominally 16 times the data rate. The DPLL uses this clock, along with the BPM signal, to construct a receive clock for the data. Depending on the precize timing of edges, the receive clock counter can be adjusted. To describe this mechanism accurately would require an adaptation of our model. Apart from investigating this issue, another obvious direction for future research is to carry out a similar analysis for the Manchester encoding protocol as it is used in e.g. the Ethernet.

UPPAAL has turned out to be an (almost) perfect tool for this type of application. Modelling the biphase mark protocol in terms of networks of timed automata is very natural, the graphical user interface helps to visualize the automata, the simulator is a great help during the initial validation of the model, and the ability of UPPAAL to generate counterexamples and to replay them in the simulator greatly helped to increase our insight in the protocol.

Several authors have explored extensions of timed automata tools that are able to handle parametrized timed automata and to verify/synthesize parameter constraints [HRSV02, AAB00, CS01]. We have arrived at the conclusion that this is probably not the way to go. Adding the feature of parameter handling to model checkers greatly affects performance and reduces the size of the systems that can be handled. Still, generation of nonlinear constraints (like in the case of BMP) turns out to be difficult. The ability to handle complex models is essential for the success of model checking technology. The protocol discussed in the present article is very simple, but even in this case adding additional features such as termination, bus collisions, and a more accurate modelling of the hardware would probably push UPPAAL to its limits. In our experience, it is typically easy to come up with general parameter constraints (linear or nonlinear) based on the counterexamples produced by UPPAAL. The challenge therefore is to verify correctness of the parametrized system while assuming these constraints. For this, the most promising approach in our view is via a translation of the UPPAAL model to a general purpose theorem prover such as PVS and exploitation of powerful invariant generation methods such as the ones proposed by [BLS96, BBF$^+$00]. For practical reasons and also because we had already a manual proof of the invariants available, we just used PVS and not any of the additional invariant generation methods.

The proof of the correctness of the biphase mark protocol in PVS is required since the collection of UPPAAL invariants alone is not enough to establish that the parameter constraints are necessary and sufficient for the correctness of the protocol. The formalization in PVS revealed a small omission in the invariants from the manual proof and enabled us to establish global correctness of the protocol for all of its instances. Additionally, we show how a small effort in the automation of proofs can produce great improvements in proof size and readability.

Acknowledgments

The authors would like to thank the Stan Ivanov, participants to System Modelling Course for Philips Research, Jozef Hooman, Martijn Hendriks and Maarten Boasson for their comments on earlier version of this paper and the formalization of the biphase mark protocol.

References

[AAB00] Annichini A, Asarin E, Bouajjani A (2000) Symbolic techniques for parametric reasoning about counter and clock systems. In: Emerson EA, Sistla AP (eds) Proceedings of the 12th international conference on Computer aided verification, vol 1855 of *Lecture Notes in Computer Science*. Springer, Berlin Heidelberg New York, pp 419–434

[AD94] Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126:183–235

[Adv92] Advanced Micro Devices, Inc (1992) Technical Manual Am8530H/Am85C30 Serial Communications Controller

[AH96] Alur R, Henzinger TA (eds) (1996) Proceedings of the 8th international conference on computer aided verification, New Brunswick, NJ, USA, vol 1102 of *Lecture Notes in Computer Science*. Springer, Berlin Heidelberg New York

[AHS98] Myla Archer (1998) Constance Heitmeyer, and Steve Sims. TAME: a PVS interface to simplify proofs for automata models. In: *User interfaces for theorem Provers*. Eindhoven, The Netherlands

[BBF$^+$00] Bensalem S, Bozga M, Fernandez JC, Ghirvu L, Laknech Y (2000) A transformational approach for generating non-linear invariants. In: Palsberg J (ed) SAS, vol 1824 of *Lecture Notes in Computer Science*, Springer, Berlin Heidelberg New York, pp 58–74

[BDM$^+$98] Bozga M, Daws C, Maler O, Olivero A, Tripakis S, Yovine S (1998) Kronos: a model-checking tool for real-time systems. In: Hu AJ, Vardi MY (eds) *Proceedings of the 10th international conference on computer aided verification*, Vancouver, BC, Canada, vol 1427 of Lecture Notes in Computer Science. Springer, Berlin Heidelberg New York, pp 546–550

[BGK$^+$96] Bengtsson J, Griffioen WOD, Kristoffersen KJ, Larsen KG, Larsson F, Pettersson P, Yi W (1996) Verification of an audio protocol with bus collision using UPPAAL. In: *Alur and Henzinger [AH96]*, pp 244–256

[BLS96] Bensalem S, Lakhnech Y, Saidi H (1996) Powerful techniques for the automatic generation of invariants. In: *Alur and Henzinger [AH96]*, pp 323–335

[BM88] Boyer RS, Moore JS (1988) *A computational logic handbook*. Academic, New York

[BPV94] Bosscher DJB, Polak I, Vaandrager FW (1994) Verification of an audio control protocol. In: Langmaack H, de Roever W-P, Vytopil J (eds) *Proceedings of the third international school and symposium on formal techniques in real-time and fault-tolerant systems* (FTRTFT'94), Lübeck, Germany, September 1994, vol 863 of *Lecture Notes in Computer Science*, Springer, Berlin Heidelberg New York, pp 170–192

[CHdV03] Chkliaev D, Hooman J, de Vink E (2003) Verification and improvement of the sliding window protocol. In: *Proceedings TACAS'03, vol 2619. Lecture Notes in Computer Science*, Springer, Berlin Heidelberg New York, pp 113–127

[Cri89] Cristian F (1989) Probabilistic clock synchronization. Distributed Comput 3:146–158

[CS01] Collomb–Annichini A, Sighireanu M (2001) Parameterized reachability analysis of the IEEE 1394 Root Contention Protocol using TReX. In: Pettersson p, Yovine S (eds) *Proceedings of the Workshop on Real-Time Tools* (RT-TOOLS'2001)

[Cyr93] Cyrluk D (1993) Microprocessor verification in PVS: a methodology and simple example. *Technical Report SRI-CSL-93-12*, Computer Science Laboratory, SRI International, Menlo Park, CA

[DY95] Daws C, Yovine S (1995) Two examples of verification of multirate timed automata with KRONOS. In: *Proceedings of the 16th IEEE real-time systems symposium* (RTSS'95). Pisa, Italy. IEEE Computer Society Press, New York, pp 66–75

[Gri00] Griffioen WOD (2000) Studies in Computer Aided verification of protocols. *PhD thesis*, University of Nijmegen, May 2000. http://www.cs.kun.nl/ita/former_members/davidg/, IPA thesis no. 2000-04

[HI96] Hung DV, Kwang K II (1996) Verification via digitized models of real-time hybrid systems. In: *Proceedings Asia-Pacific software engineering conference* (APSEC'96). IEEE Computer Society Press, New York, pp 4–15

[HPWT01] Henzinger TA, Preussig J, Wong-Toi H (2001) Some lessons from the HyTech experience. In: *Proceedings of the 40th Annual Conference on Decision and Control* (CDC). IEEE Press, New York, pp 2887–2892

[HRSV02] Hune TS, Romijn JMT, Stoelinga MIA, Vaandrager FW (2002) Linear parametric model checking of timed automata. J Logic Algebraic Program 52–53:183–220

[HSLL97] Havelund K, Skou A, Larsen KG, Lund K (1997) Formal modelling and analysis of an audio/video protocol: an industrial case study using UPPAAL. In: *Proceedings of the 18th IEEE real-time systems symposium*. IEEE Computer Society Press, New York, pp 2–13

[Hun98] Hung DV (1998) Modelling and verification of biphase mark protocols using PVS. In: *Proceedings of the international conference on applications of concurrency to system design* (CSD'98), Aizu-wakamatsu, Fukushima, Japan, March 1998. IEEE Computer Society Press, pp 88–98

[HWT95] Ho P-H, Wong-Toi H (1995) Automated analysis of an audio control protocol. In: Wolper P (ed) *Proceedings of the 7th international conference on computer aided verification*, Liège, Belgium, vol 939 of *Lecture Notes in Computer Science*. Springer, Berlin Heidelberg New York

[IG99] Ivanov S, Griffioen WOD (1999) Verification of a biphase mark protocol. *Report CSI-R9915, Computing Science Institute*, University of Nijmegen, August 1999

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

[LPY97] Larsen KG, Pettersson P, Yi W (1997) UPPAAL in a Nutshell. Int J Softw Tools Technol Transf 1(1–2):134–152

[Moo94] Moore J.S[trother] (1994) A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. Formal Aspects Comput 6(1):60–91

[ORSH95] Owre S, Rushby J, Shankar N, von Henke F (1995) Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. IEEE Trans Softw Eng 21(2):107–125

[SH98] Sawada J, Hunt WA (1998) Processor verification with precise exceptions and speculative execution. In: *Proceedings of 10th international computer aided verification conference*, pp 135–146

Received 10 November 2004 by J. Rushby
Accepted 6 March 2006 by C.B. Jones
Published online 8 September 2006