Verification of a Biphase Mark Protocol

Stanimir Ivanov* and David Griffioen**

Computing Science Institute, University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{sivanov,davidg}@cs.kun.nl

Abstract We show how a symbolic model checker for linear hybrid automata can be used to analyze a biphase mark protocol. This protocol was first verified formally by Moore [Moo94] using a model of asynchrony. In this paper we demonstrate that algorithmic methods can automatically verify the correctness of the protocol for wider clock drifts. Unlike Moore [Moo94], our model allows for clock jitter. We believe that linear hybrid automata enable a natural way of modeling the protocol.
1994 Extended CR Subject Classification: B.4.4, B.6.3, C.2.2, D.2.1.
1991 AMS Subject Classification: 68Q22, 68Q60, 68Q68.
Keywords and Phrases: Formal Methods, Verification, Model Checking, Hybrid Automata, Communication Protocols, Biphase Mark.

1 Introduction

The goal of this paper is to prove the correctness of a biphase mark protocol within the model of linear hybrid automata [AHH93]. Biphase mark [Moo94] was first analyzed by Moore using a model of asynchrony. The correctness was checked using the Boyer-Moore theorem prover, Nqthm [BM88]. In this paper, we consider the following questions:

- How natural is the modeling of biphase mark in the formalism of linear hybrid automata and how complex are the correctness proof of the protocol in that formalism?
- Is it possible to generalize or improve Moore's results [Moo94]?

The biphase mark protocol is a convention for representing both a string of bits and clock edges in a square wave [Moo94]. Biphase mark is widely used in applications where data written by one device is read by another. It is an industry standard for single density magnetic floppy disc recording and is one of several protocols implemented by commercially available microcontrollers such as Intel 82530 Serial Communications Controller [Cor91]. A version of biphase

* The work on this paper was carried out in partial fulfillment of the requirements for the Master's degree in Mathematics and Computer Science at the University of Nijmegen. The thesis was written under the guidance of prof.dr. F.W. Vaandrager.
** Supported by the Netherlands Organization for Scientific Research (NWO) under contract SION 612-316-125.

mark, called "Manchester", is used in the Ethernet [Rod88] and is implemented in the Intel 82C501AD Ethernet Serial Interface [Cor91].

We describe the biphase mark protocol using linear hybrid automata [ACHH93]. These automata model nondeterministic continuous activities of analog variables, as well as discrete events. The state of the automata changes either through instantaneous system actions or, while time elapses, through differentiable environment activities. A hybrid system is described as a collection of hybrid automata, one per component, that operate concurrently and synchronize with each other. The communication is achieved via shared variables as well as synchronization labels. Model-checking based analysis techniques [ACHH93] have been implemented in HyTech [HHWT95] and have been used to verify various components of embedded systems [HWT96,HHWT95,HWT95].

In Section 2 we describe the individual components of the digital transmission system under consideration and give an informal description of the biphase mark protocol. In Section 3 we present our system modeling language: linear hybrid automata. In Section 4 the biphase mark protocol is modeled as a parallel composition of three linear hybrid automata and we discuss how the correctness criteria have been automatically verified using the symbolic model checker HyTech.

The two main results established in this paper are:

- Linear hybrid automata enable a natural way of modeling the biphase mark protocol.
- HyTech easily verified the correctness of biphase mark for wider clock drifts than those given in [Moo94].

However, we also we came across certain limitations of HyTech, concerning parametric analysis and some values of the parameters of the clock drift.

2 Informal Description of the Biphase Mark Protocol

In this section the individual components of the digital transmission system under consideration are described. Fig. 1 shows the block diagram of the transmission system. It consists of a *digital source*, a *sender*, a *transmission channel*, a *receiver*, and a *digital sink*. The digital source generates a *bit sequence* $a = (a_1, \ldots, a_k)$ of length k, which is physically represented by the *signal* $s(t)$ and is to be transmitted to a digital sink. The transmitted signal $s(t)$ is distorted during transmission via the channel. The distorted signal at the end of the transmission channel is denoted by $d(t)$. An error-free transmission results if the *sink bit sequence* $b = (b_1, \ldots, b_l)$ is identical with the source bit sequence.

2.1 Digital Source

The digital source repeatedly generates a bit sequence $a = (a_1, \ldots, a_k)$ of varying length $k \geq 0$, which is kept in a list. The sender has access only to the first element of the sequence. While bit a_i is being transmitted, the digital source

Figure1. Block diagram of a digital transmission system

makes sure that a_i is removed from the list in order to provide access to the next bit in the sequence in the case $i < k$. In the case $i = k$, the sender can observe that the last bit of the sequence was sent and stops transmitting until the next bit sequence is generated. Notice that the generated bit sequences need not be of the same length.

The amount of time which is available for the transmission of one bit is defined as the *bit duration* T. We refer to $1/T$ as the *bit rate*, the unit of which is bit per second.

2.2 Clock Signals

The sender and the receiver are digital clocked devices. By a *clock signal* we mean a *periodic rectangular signal* $z(t)$ as represented in Fig. 2(a). The begin of each clock cycle is fixed by the leading edge of the signal. The length between two neighboring leading edges is referred as to a *clock cycle*. In general it has to be assumed, however, that the leading edges do not follow each other equidistantly, and thus the clock cycles are not equal. This is then referred to as a jittering clock signal. Fig. 2(b) shows an exaggerated example of a jittering clock. In [Moo94] the following assumption is made about the clock signal:

> *The clocks of both processors are linear functions of real time, e.g., the ticks of a given clock are equally spaced events in real time. We ignore clock jitter.*

In this paper we do consider the more general jittering clock signals.

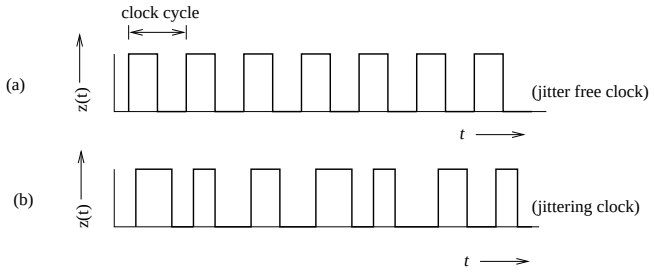

Figure2. A graph of clock signals

2.3 Sender

The sender encodes each bit a_i in a *cell* according to a modulation technique known as the *biphase mark protocol* [Moo94] (see Fig. 3). The biphase mark protocol is a convention for representing both a sequence of bits and clock edges in a square wave.

Define the *cell size* of a cell as the number of clock cycles available for the encoding of one bit. Each cell is then logically divided into a *mark subcell*, consisting of *mark size* clock cycles, followed by a *code subcell*, consisting of *code size* clock cycles.

During the mark subcell the waveform is held at the negation of its value at the end of the previous cell, providing an edge in the signal train which marks the beginning of the new cell. During the code subcell, the signal either returns to its previous value or not, depending on whether the cell encodes a 1 or a 0, respectively.

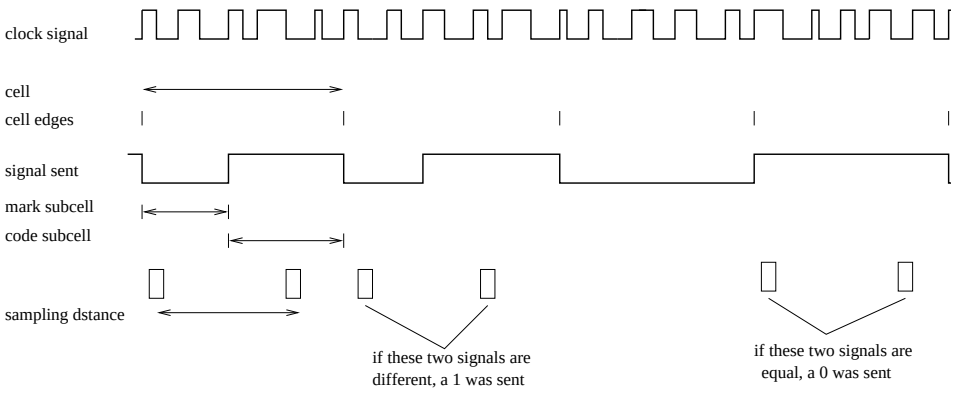

Figure3. Biphase mark terminology

2.4 Transmission Channel

The transmission channel encompasses all devices which lie between the sender and the receiver. Its main component is the transmission medium which may be, for example, a symmetric pair cable, a coaxial cable or an optical waveguide.

The transmission characteristics of the channel are time, frequency, amplitude and temperature dependent. Therefore the transmitted signal is distorted during transmission via the channel.

Let $s(t)$ denote the channel input as a function of time; $s(t)$ could represent a voltage or a current waveform. Similarly, let $d(t)$ represent (the voltage or the current waveform at) the output of the channel. The output $d(t)$ is called the

detection signal. It is a distorted, delayed and attenuated version of $s(t)$. One of the most distorting effects in most transmission channels is linear time-invariant filtering [BG87]. Filtering occurs not only from filters inserted by the channel designer but also from inherent behavior of the propagation medium. One effect of filtering is to "smooth out" the transmitted signal $s(t)$ (see Fig. 4).

The following assumption about the output of the detection signal $d(t)$ is made in [Moo94]:

> *The distortion in the signal $d(t)$ due to presence of an edge is limited to the time-span of the cycle during which the edge was written. For example we ignore intersymbol interference [Rod88].*

Figure4. The relation between input and output waveforms for transmission channel with filtering

2.5 Receiver

The digital receiver extracts information from the detection signal $d(t)$ using a threshold device for amplitude regeneration and a clock signal $v(t)$ for determining the *sample time* at which the receiver samples its input signal. Analogous to the sender's clock we assume that the receiver's clock signal also jitters. Thus, the sample times t_i do not follow each other equidistantly. The operation of the threshold device can be described using its threshold value E. If the detection signal $d(t_i)$ at time t_i is greater than the threshold value, the sample value is set equal to 1. However, if $d(t_i) \leq E$ then the sample value is set equal to 0. Notice that the use of a threshold device is equivalent with the assumption of [Moo94]:

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

The receiver is generally waiting for the edge that marks the arrival of a cell. The edge is detected by sampling the detection signal $d(t)$ at the end of each clock cycle. When the receiver samples a value of $d(t)$ which is different from the previous sample value, it detects an edge in the signal train. Upon detecting the edge, the receiver counts off a fixed number of cycles, called the *sampling distance*, and samples the signal there. The sampling distance is determined so as to make the receiver sample approximately in the middle of the code subcell. If the sample is the same as the mark, then a 0 was sent; otherwise a 1 was

sent. The receiver then resumes waiting for the next edge, thus "phase locking" into the sender's clock. Phase locking is the activity of adjusting the clocks of two or more processes so that all clocks tick "simultaneously". A common technique is for the sender to encode its clock in the signal stream and for the receiver to adjust its timing accordingly. Phase locking is often done with special device that changes the rate at which crystals vibrate. But by adopting an artificially slow "virtual" clock, e.g., where one virtual tick occurs every n physical ticks, it is possible to implement phase locking in software or firmware. This is called "digital phase locking". Biphase mark protocols are often used in such implementations [Moo94].

We adopt the notation $bpm(n, m, l)$ (abbreviation for *biphase mark*(n, m, l)) for the version of the protocol with cell size equal to n, mark size equal to m, sampling distance equal to l. Examples of such configurations are $bpm(18, 5, 10)$, $bpm(16, 8, 11)$, and $bpm(32, 16, 23)$. The $bpm(16, 8, 11)$ configuration is implemented in the Intel 82530 Serial Communications Controller [Cor91]. The model of Moore [Moo94] was not powerful enough to verify this configuration. In [Moo94] the configurations $bpm(16, 8, 11)$ and $bpm(32, 16, 23)$ are verified for error tolerances of $1/32$ and $1/18$ respectively.

3 System Modeling Language

In this section we present our system modeling language. We closely follow in content and presentation the work of R. Alur, T.A. Henzinger, and P.-H. Ho [AHH93] and the user guide for HyTech [HHWT95].

3.1 Linear Hybrid Automata

Informally, a linear hybrid automaton [ACHH93] consist of a finite vector $\boldsymbol{x}$ of real-valued variables and a labeled multigraph (V, E). The edges from E represents discrete system actions and are labeled with constraints on the values of $\boldsymbol{x}$ before and after actions. The vertices of V represent continuous environment activities and are labeled with constraints on the values and first derivatives of $\boldsymbol{x}$ during activities. The state of the automaton changes either through instantaneous system actions or, while time elapses, through differentiable environment activities. We restrict ourselves to edge and vertex constraints that are linear expressions. A hybrid system is described as a collection of hybrid automata, one per component, that operate concurrently and communicate with each other. Communication is achieved via shared variables as well as synchronization labels.

We use a simple railroad crossing [HHWT95] as a running example. The system consists of three components: a train, a gate, and a controller. The train is initially some distance — at least 2000 meter — away from the track intersection with the gate fully raised. As the train approaches, it triggers a sensor — 1000 meter ahead of the intersection — signaling its upcoming entry to the controller. The controller then sends a lower command to the gate, after a delay of up to

α seconds. When the gate receives a lower command, it lowers at rate of 9 degrees per second. After the train has left the intersection and is 100 meter away, another sensor sends an exit signal to the controller. The controller then commands the gate to be raised. The role of the controller is to ensure that the gate is always closed whenever the train is in the intersection, and the gate is not closed unnecessarily long. The linear hybrid automata for the train, the gate, and the controller appear in Figs. 5, 6, and 7.

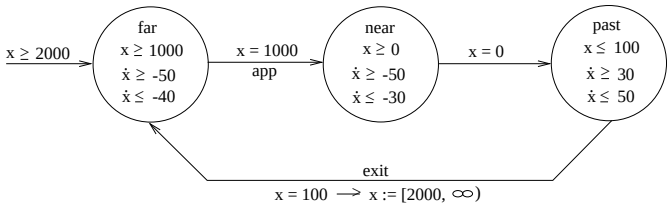

Figure5. Train automaton

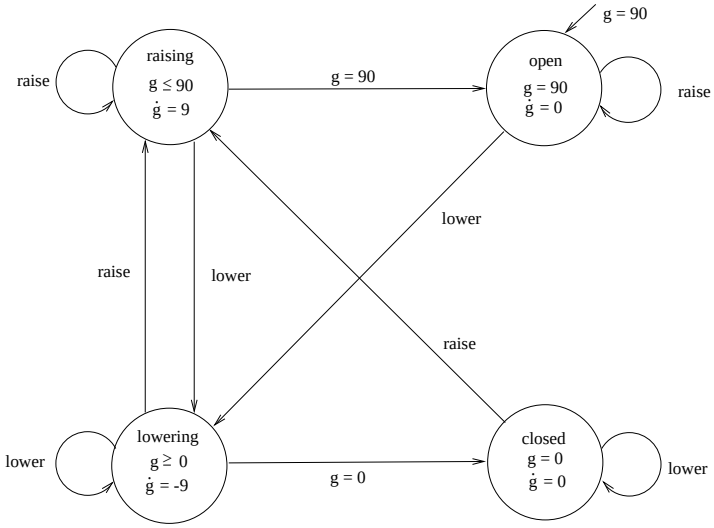

Figure6. Gate automaton

Syntax. Let $\boldsymbol{u}$ be a vector of real-valued variables. A *linear term* over $\boldsymbol{u}$ is a linear combination of variables from $\boldsymbol{u}$ with integer coefficients. A *linear inequal-*

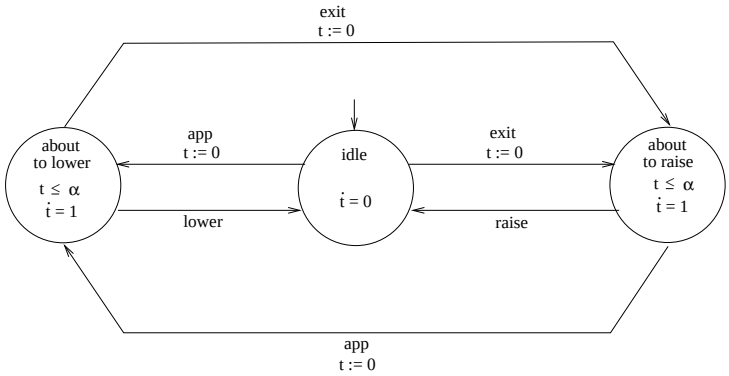

Figure7. Controller automaton

ity over $\boldsymbol{u}$ is an inequality between linear terms over $\boldsymbol{u}$. A *(closed) convex linear formula* over $\boldsymbol{u}$ is a finite conjunction of (nonstrict) linear inequalities over $\boldsymbol{u}$. A *linear formula* over $\boldsymbol{u}$ is a finite boolean combination of linear inequalities over $\boldsymbol{u}$. Every linear formula can be transformed into disjunctive normal form, that is, into a finite disjunction of convex linear formulas.

A *linear hybrid automaton* $A = (\boldsymbol{x}, V, \varphi^0, \mathit{inv}, \mathit{dif}, E, \mathit{act}, L, \mathit{syn})$ consists of the following components:

Data variables A finite vector $\boldsymbol{x} = (x_1, \ldots, x_n)$ of real-valued *data variables*. For example, the position of the train is determined by the value of the variable x, which represents the distance of the train from the intersection. The variable g models the angle of the gate. When $g = 90$, the gate is completely open; when $g = 0$, it is completely closed.

A *data state* is a point $\boldsymbol{s} = (s_1, \ldots, s_n)$ in the n-dimensional real space $\mathbb{R}^n$ or, equivalently, a function that assigns to each data variable x_i a real value $s_i \in \mathbb{R}$. A *convex data region* is a convex polyhedron in $\mathbb{R}^n$. A *data region* is a finite union of convex data regions. A *(convex) data predicate* is a (convex) linear formula over $\boldsymbol{x}$, e.g., $x_1 \geq 3 \wedge 3x_2 \leq x_3 + 5/2$. The (convex) data predicate p defines a (convex) data region $[\![p]\!] \subseteq \mathbb{R}^n$, where $\boldsymbol{s} \in [\![p]\!]$ iff $p[\boldsymbol{x} := \boldsymbol{s}]$ is true. A *differential inclusion* is a convex polyhedron in $\mathbb{R}^n$. A *rate predicate* is a convex linear formula over the vector $\dot{\boldsymbol{x}}(\dot{x}_1, \ldots, \dot{x}_n)$ of dotted variables. The rate predicate r defines the differential inclusion $[\![r]\!] \subseteq \mathbb{R}^n$, where $\dot{\boldsymbol{s}} \in [\![r]\!]$ iff $r[\dot{\boldsymbol{x}} := \dot{\boldsymbol{s}}]$ is true. For each data variable x_i we use the primed variables x_i' to denote the new value of x_i after a transition. An *action predicate* $q = (\boldsymbol{y}, q')$ consists of a set $\boldsymbol{y} \subseteq \boldsymbol{x}$ of *updated variables* and a convex linear formula q' over the set $\boldsymbol{x} \uplus \boldsymbol{y'}$ of data variables and primed updated variables. The *closure* $\{q\}$ of the action predicate q is the convex linear formula $q' \wedge \bigwedge_{x \in \boldsymbol{x} \setminus \boldsymbol{y}} (x' = x)$; that is, all data variables that are not updated remain unchanged. The action predicate q defines a function $[\![q]\!]$ from data states to convex data regions: for all data states $\boldsymbol{s}, \boldsymbol{s'} \in \mathbb{R}^n$,

let $\boldsymbol{s}' \in [\![q]\!](\boldsymbol{s})$ iff $\{q\}[\boldsymbol{x}, \boldsymbol{x}' := \boldsymbol{s}, \boldsymbol{s}']$ is true. The action predicate q is *enabled* in the data state $\boldsymbol{s}$ if the data region $[\![q]\!](\boldsymbol{s})$ is nonempty.

Control locations A finite set V of vertices called *control locations*. For example, the gate automaton in Fig. 6 has the locations *open, rising, lowering*, and *closed*. A *state* $(v, \boldsymbol{s})$ of the automaton A consists of a control location $v \in V$ and a data state $\boldsymbol{s} \in \mathbb{R}^n$. A *region* $R = \bigcup_{v \in V}(v, S_v)$ is a collection of data regions $S_v \subseteq \mathbb{R}^n$, one for each location $v \in V$. A *state predicate* $\phi = \bigcup_{v \in V}(v, p_v)$ is a collection of data predicates p_v, one for each location $v \in V$. The state predicate ϕ defines the region $[\![\phi]\!] = \bigcup_{v \in V}(v, [\![p_v]\!])$. The region R is *linear* if there is a state predicate that defines R. We write (v, S) for the region $(v, S) \cup \bigcup_{v' \neq v}(v', \emptyset)$, and (v, p) for the state predicate $(v, p) \cup \bigcup_{v' \neq v}(v', \mathit{false})$. When writing state predicates, we use the *location counter* l, which ranges over the set V of control locations. The location counter $l = v$ denotes the state predicate (v, true). The data predicate p, when used as a state predicate, denotes the collection $\bigcup_{v \in V}(v, p)$. For two state predicates $\phi = \bigcup_{v \in V}(v, p_v)$ and $\phi' = \bigcup_{v \in V}(v, p'_v)$, we define $\neg\phi = \bigcup_{v \in V}(v, \neg p_v)$, $(\phi \vee \phi') = \bigcup_{v \in V}(v, p_v \vee p'_v)$ and $(\phi \wedge \phi') = \bigcup_{v \in V}(v, p_v \wedge p'_v)$.

Initial condition A state predicate φ^0 called the *initial condition*. For example, the gate is initially in location *open* with the value of g equal to 90. In graphical representations, if φ^0 is of the form (v^0, p^0), a small incoming arrow, labeled with p^0, identifies the initial location v^0.

Location invariants A labeling function *inv* that assigns to each control location $v \in V$ a convex data predicate $\mathit{inv}(v)$, the *invariant* of v. The invariants are used to enforce the progress of a system from one control location to another, because the control of the automaton A may reside in the location v only as long as the invariant $\mathit{inv}(v)$ is true. For example, in the gate automaton, $\mathit{inv}(\mathit{open}) = (g = 90)$, $\mathit{inv}(\mathit{lowering}) = (g \geq 0)$, $\mathit{inv}(\mathit{raising}) = (g \leq 90)$, and $\mathit{inv}(\mathit{closed}) = (g = 0)$. The invariant for location *lowering* ensures that the gate is lowered until it is fully closed, at which point control has to move to location *closed*. In the graphical representation, an invariant *true* is omitted. The state $(v, \boldsymbol{s})$ is *addmissible* if $\boldsymbol{s} \in [\mathit{inv}(v)]$. We write Σ_A for the admissible states of A, and ϕ_A for the state predicate $\bigcup_{v \in V}(v, \mathit{inv}(v))$ that defines the set of admissible states.

Continuous activities A labeling function *dif* that assignes to each control location $v \in V$ a rate predicate $\mathit{dif}(v)$, the *activity* of v. The activities contain the rates at which the values of data variables may change. While the automaton control resides in the location v, the first derivatives of all data variables stay within the differential inclusion $[\![\mathit{dif}(v)]\!]$. In the gate automaton, the rate predicate for locations *open* and *closed* is $\dot{g} = 0$, for location *raising* it is $\dot{g} = 9$, and for *lowering* it is $\dot{g} = -9$.

Transitions A finite multiset E of edges called *transitions*. Each transition (v, v') identifies a source location $v \in V$ and a target location $v' \in V$. For example, the train automaton has three transitions; one from location *far* to location *near* for entering the region immediately surrounding the intersection, one from *near* to *past* for going through the intersection, and

one from *past* to *far* for exiting the region around the intersection. Transitions may optionally be assigned the *urgent flag* ASAP. Transitions so labelled are called *urgent*

In addition, there is for each location $v \in V$, a *stutter transition* $e_v = (v, v)$.

Discrete actions A labeling function act that assigns to each transition $e \in E$ an action predicate $act(e)$, the *action* of e. Control can proceed from a location v to a location v' via the transition $e = (v, v')$ only when the action $act(e)$ is enabled. If $act(e)$ is enabled in the data state $\boldsymbol{s}$, then the value of the data variables may change nondeterministically from $\boldsymbol{s}$ to some point in the data region $[\![act(e)]\!](\boldsymbol{s})$. The action predicates can be used for synchronizing hybrid automata via shared variables. All stutter transitions are labeled with the action $(\boldsymbol{x}, \boldsymbol{x'} = \boldsymbol{x})$, and thus, leave all data variables unchanged.

Synchronization labels/function A finite set L of *synchronization labels* and a labeling function *syn* that assigns to each transition $e \in E$ a set of synchronization labels from L. The synchronisation labels are used to define the parallel composition of two automa. If both automata share a synchronization label a, then each a-transition (that is, a transition whose syncronization label contains a) of one automaton must be accompained by an a-transition of the other automaton. All stutter transitions are labeled with the empty set of transition labels. For example, in the gate automaton, the transition from *open* to *lowering* has the synchronization label *lower*, and this synchronizes (i.e., must be taken simultaneously) with the transition labeled *lower* in the controller automaton. In graphical notations we often write a for the singelton set $\{a\}$.

Semantics. At any time instant, the state of a hybrid automaton specifies a control location and values for all data variables. The state can change in two ways: (1) by an instantaneous transition that may change both the control location and the values of data variables, or (2) by a time delay that may change only the values of data variables in continuous manner according to the rate predicate of the current control location.

A *data trajectory* (δ, ρ) of the linear hybrid automata A consists of a nonnegative *duration* $\delta \in \mathbb{R}^{\geq 0}$ and a differentiable function $\rho : [0, \delta] \to \mathbb{R}^n$ with the derivative $\frac{d\rho(t)}{dt}$ for all $t \in (0, \delta)$. The data trajectory (δ, ρ) maps every real $t \in [0, \delta]$ to a data state $\rho(t)$.

A data trajectory (δ, ρ) is a v-trajectory, for a location $v \in V$, if (1) for all reals $t \in [0, \delta], \rho(t) \in [\![inv(v)]\!]$, and (2) for all reals $t \in (0, \delta)$, $\frac{d\rho(t)}{dt} \in [\![dif(v)]\!]$. A *trajectory* τ of A is a infinite sequence

$$(v_0, \delta_0, \rho_0) \to (v_1, \delta_1, \rho_1) \to (v_2, \delta_2, \rho_2) \to (v_3, \delta_3, \rho_3) \to \ldots \ .$$

of control locations $v_i \in V$ and corresponding v_i-trajectories (δ_i, ρ_i) such that for all $i \geq 0$, there is a transition $e_i = (v_i, v_{i+1}) \in E$ with $\rho_{i+1}(0) \in [\![act(e_i)]\!](\rho_i(\delta_i))$.

A *position* of the trajectory τ is a pair (i, ϵ) that consists of a nonnegative integer i and a nonnegative real $\epsilon \leq \delta_i$. The position of τ are ordered lexicographically: the position (i, δ) precedes the position (j, ϵ), denoted $(i, \delta) <$

(j, ϵ), iff either $i < j$, or $i = j$ and $\delta < \epsilon$. The *state at position* (i, ϵ) of τ is $\tau(i, \epsilon) = (v_i, \rho_i(\epsilon))$. The *time at position* (i, ϵ) of τ is the finite sum $t_\tau(i, \epsilon) = \left(\sum_{0 \le j < i} \delta_j\right) + \epsilon$. The *duration* of the trajectory τ is the infinite sum $\delta_\tau = \sum_{j \ge 0} \delta_j$.

Parallel Composition. A hybrid system typically consists of several components that operate concurrently and communicate with each other. We describe each component as a linear hybrid automaton. The component automata coordinate through shared data variables and synchronization labels. The linear hybrid automaton that models the entire system is then constructed from the component automata using a product operation.

Let $A_1 = (\boldsymbol{x_1}, V_1, \varphi_1^0, inv_1, dif_1, E_1, act_1, L_1, syn_1)$ and $A_2 = (\boldsymbol{x_2}, V_2, \varphi_2^0, inv_2, dif_2, E_2, act_2, L_2, syn_2)$ be two linear hybrid automata. The *product* $A_1 \times A_2$ of A_1 and A_2 is the linear hybrid automaton $A = (\boldsymbol{x_1} \times \boldsymbol{x_2}, V_1 \times V_2, \varphi_1^0 \wedge \varphi_1^0, inv, dif, E, act, L_1 \cup L_2, syn)$, where

- Each location (v, v') in $V_1 \times V_2$ has the invariant $inv(v, v') = inv_1(v) \wedge inv_2(v')$ and the activity $dif(v, v') = dif_1(v) \wedge dif_2(v')$. Thus, an admissible state of A consists of an admissible state of A_1 and an admissible state of A_2, whose parts coincide, and whose rate vectors obey the differential inclusions that are associated with both components locations.
- E contains the transition $e = ((v_1, v_1'), (v_2, v_2'))$ iff
  1. there is a transition $e_2 = (v_2, v_2') \in E_2$ with $syn(e_2) \cap L_1 = \emptyset$ and $v_1 = v_1'$; or
  2. there is a transition $e_1 = (v_1, v_1') \in E_1$ with $syn(e_1) \cap L_2 = \emptyset$ and $v_2 = v_2'$; or
  3. there is a transition $e_1 = (v_1, v_1') \in E_1$ and a transition $e_2 = (v_2, v_2') \in E_2$ such that $syn(e_2) \cap L_1 = syn(e_1) \cap L_2$.

  In case (1), $act(e) = act_2(e_2)$ and $syn(e) = syn_2(e_2)$. In case (2), $act(e) = act_1(e_1)$ and $syn(e) = syn_1(e_1)$ In case (3), if $act_1(e_1) = (\boldsymbol{y}_1, q_1')$ and $act_2(e_2) = (\boldsymbol{y}_2, q_2')$, then $act(e) = (\boldsymbol{y}_1 \cup \boldsymbol{y}_2, q_1' \wedge q_2')$ and $syn(e) = syn_1(e_1) \cup syn_2(e_2)$.

According to the definition of E, the transitions of the two component automata are interleaved, provided they have no labels in $L_1 \cap L_2$. Labels in $L_1 \cap L_2$ may be synchronized, and cause the simultaneous traversal of component transitions. When two transitions are synchronized, the set of updated variables of the component transitions are joined together and the constraints on the updated values are obtained by taking the conjunction of constraints imposed by the component transitions. This explains the role of the updated variables in the action predicates. In case (3), the stutter transitions of the component automata result in stutter transition of the product automaton.

3.2 Reachability and Safety Verification

At any time instant, the state of hybrid automaton specifies a location and values of all variables. If the hybrid automaton has location set V and n variables, then

the state space is defined as $V \times \mathbb{R}^n$. We define the binary *transition-step* relation, $\xrightarrow{\sigma}$, over admissible states such that $(v, \boldsymbol{s}) \xrightarrow{\sigma} (v', \boldsymbol{s'})$ iff the state $(v', \boldsymbol{s'})$ can be reached from the state $(v, \boldsymbol{s})$ by taking a transition. We assume that for every urgent transition u, if $(v, \boldsymbol{s}) \xrightarrow{u} (v', \boldsymbol{s'})$, then for all valuations $\boldsymbol{s_0}$ satisfying $inv(v)$ there exists a valuation $\boldsymbol{s'_0}$ such that $(v, \boldsymbol{s_0}) \xrightarrow{u} (v', \boldsymbol{s'_0})$. A location v is called urgent if there exists a valuation $\boldsymbol{s}$ and an urgent transition u, such that u is enabled at $(v, \boldsymbol{s})$. No time is allowed to pass in such a location. Next we define the *time-step* relation, $\xrightarrow{\tau}$, such that $(v, \boldsymbol{s}) \xrightarrow{\tau} (v', \boldsymbol{s'})$ iff $v = v'$, and there exists a real $\delta \geq 0$ such that $\delta > 0$ implies v is not urgent, and there is a function $f : [0, \delta] \rightarrow \mathbb{R}^n$ such that (1) $f(0) = s$, (2) $f(\delta) = s'$, (3) for all $t \in [0, \delta]$, $f(t)$ satisfies$inv(v)$, and (4) for all time $t \in (0, \delta)(df_1(t)/dt, (df_2(t)/dt, \ldots (df_n(t)/dt)$ satisfies $act(v)$, where $f_i(t)$ denotes the value of the ith component of the vector $\boldsymbol{x}$ in the valuation $f(t)$. We now define the binary successor relation $\rightarrow_A$ over states as $\xrightarrow{\tau} \cup \xrightarrow{\sigma}$. For a region W, we define $post(W)$ to be the set of all successor states of W, i.e., all states reachable from a state in W via a single transition or time step. The region *forward reachable* from W is defined as the set of all states reachable from W after a finite number of steps, i.e., the infinite union $post^*(W) = \bigcup_{i \geq 0} post^i(W)$. Similarly, we define $pre(W)$ to be the set of all predecessor states of W, and we let the region *backward reachable* from W be the infinite union $pre^*(W) = \bigcup_{i \geq 0} pre^i(W)$.

In practice, many problems to be analyzed can be posed in natural way as reachability problems. Often, the system is composed with a special monitor process that "watches" the system and enters a violation state whenever the execution violates a given safety requirement. Indeed all timed safety requirements [Hen92,AHH93], including bounded-time response requirements, can be verified in this way. A state (v, s) is *initial* if v is the initial location, and s satisfies the initial predicate. A system with initial states I is correct with respect to violation states Y iff $post^*(I) \cap Y = \emptyset$, or equivalently iff $pre^*(Y) \cap I$ is empty.

HyTech computes the forward reachable regions by finding the limit of the infinite sequence $I, post(I), post^2(i), \ldots$ of regions. Analogously, the backward reachable region is found by iterating pre. These iteration schemes are semidecision procedures: there is no guarantee of termination.

4 Verification of the Biphase Mark Protocol

4.1 System Description

The system to be verified is modeled as the composition of three linear hybrid automata. Figure 8 shows a flow-graph of the automata. The nodes represents the automata and the edges represents the communication between the automata. Figures 9, 10, and 11 show a graphical representation of these automata. We use the configuration $bpm(18, 5, 10)$,that is, cell size 18, mark size 5, and sampling distance 10, as a running example. The test automaton in Fig. 9 generates nondeterministically one bit, say a_i, makes a request to the sender to transmit a_i, checks if the receiver has received it correctly, generates the following bit a_{i+1}, and so on till it chooses not to generate a new bit. In this way a bit sequence

$(a_1, \ldots, a_k)$ is built up. Upon arrival of a request the sender automaton transmits a signal according to the biphase mark protocol. If there is no request, the signal is kept constant. The receiver also operates according to the biphase mark protocol. Upon receiving a bit the receiver automaton passes that bit to the test process.

Our model is based on the following assumptions.

- The distortion in the detection signal $d(t)$ due to presence of an edge in the signal $s(t)$ is limited to the time-span of the sender' clock cycle during which the edge was written [Moo94].
- Reading on an edge, that is, reading during a sender's clock cycle when the signal $s(t)$ changes its value, produces nondeterministically defined signal values, not indeterminate values [Moo94].
- The clock signals of the sender and the receiver are independent and jittering (see Sect. 2.2).

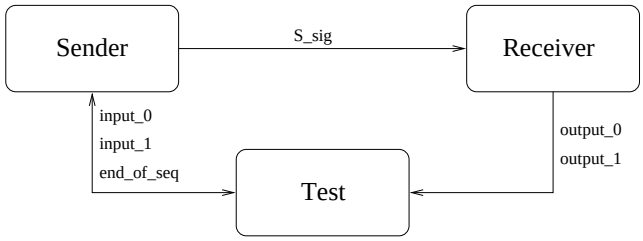

Figure8. Flow-graph of the model representing the automata and the communication between them

Transmission Channel. In order to simplify our system, we assume that the signal $s(t)$ travels with negligible delay. Hence, we can model the transmission channel as a discrete variable S_sig with domain $\{0, 1, 2\}$, representing the amplitude of the signal $d(t)$. Let $S\_\text{sig}(t)$ denotes the value of S_sig at time t. We define:

$$S\_\text{sig}(t) = \begin{cases} 0 & \text{if } d(t) \text{ is stable and low,} \\ 1 & \text{if } d(t) \text{ is stable and high,} \\ 2 & \text{otherwise.} \end{cases}$$

In other words, we define $S\_\text{sig}(t) = 0$ ($S\_\text{sig}(t) = 1$) if the receiver is able to determine deterministically the value of the signal $s(t)$ as low (high), that is, after comparing the sample value $d(t)$ to the threshold value E, then the sample value is set equal to 0 (1) (see Sect. 2.5). If this is not the case, we define

$S\_\mathrm{sig}(t) = 2$, which means that it is not always the case that $d(t) \geq E$ iff $s(t) \geq E$. In our model the sample value is then nondeterministically chosen. Notice that $S\_\mathrm{sig}(t) = 2$ when the signal $s(t)$ changes its value, according to our assumptions. In our model the sender automaton only writes into the variable S_sig and the receiver automaton only reads from the variable S_sig.

Digital Source and Digital Sink. The digital source together with the digital sink are modeled by the test automaton in Fig. 9. Initially the test automaton is in the *OK* location. Assuming the sender– and the receiver–automata in Figs. 10 and 11, the test automaton generates nondeterministically the source bit sequence $(a_1, \ldots, a_k)$, builds the sink bit sequence $(b_1, \ldots, b_k)$ from the information passed from the receiver automaton, and checks if $a_i = b_i$ for each $i = 1, \ldots, k$, and $k \in \mathbb{N}$. This is easily proved by induction on the length of the source sequence.

If $k = 0$ then the test automaton can choose to stay in the *OK* location by taking the action labeled with synchronization label *end_of_seq*, which indicates the end of a bit sequence, hence the empty sequence is then generated. Assume for the induction hypothesis that $a_i = b_i$ for $i = 1, \ldots k-1$. Assume further that the test automaton is in the *OK* location. At the end of a bit cell, i.e., sender's clock x is equal to 18, the test automaton makes a move, say, to the *I1* location. At the same time it makes a request to the sender automaton to transmit the information bit 1, modeled by the synchronization label *input_1*. Thus $a_k = 1$. It stays in the *I1* location till the receiver automaton passes the received information bit by means of the synchronization labels *output_0* or *output_1*. In the first case the receiver has received a bit 0, that is $b_k = 0$ and the test automaton moves to the *error* location. In the second case $b_k = 1$ and the test automaton moves to the *OK* location. The *error* location can also be reached from the *I1* location via the actions labeled with the *input_1*, *input_0*, or *end_of_seq* synchronization labels. This is the case when the sender automaton has completed the transmission of one single bit, but the receiver automaton has not yet received it. The case when the test automaton moves to the *I0* location is similar.

Sender. The sender is modeled by the sender automaton in Fig. 10. See also Appendix A for the HyTech code. Initially the sender automaton is in the *new_cell* location and the value of the sender's clock x is between 17 and 18 modeling the phase difference between the sender's and the receiver's clocks, and the value of S_sig is equal to 1. The auxiliary variable S_prev with a domain $\{0, 1\}$ is used to remember the last value of S_sig before it is changed to 2. The initial value of S_prev is equal to that of S_sig. Only in the *new_cell* location the sender accepts a request to send a message bit. This choice results in a sender that is not input enabled according to the theory of I/O-automata model [LT89]. The sender in fact dictate the environment by saying when it is ready to transmit an information bit. The reason for this choice is the complexity of the test automaton, which is in fact "one bit buffer". The input request is modeled

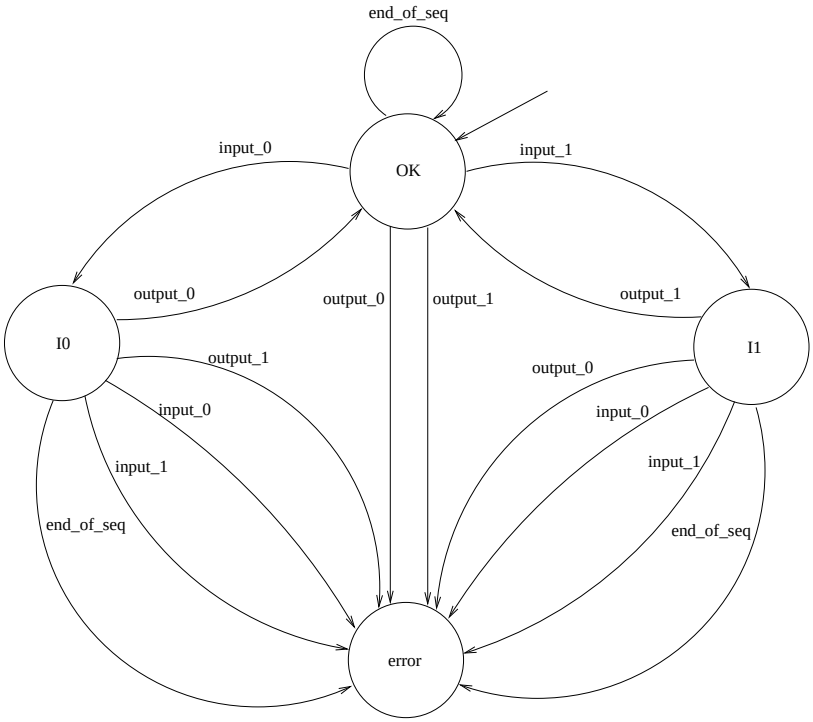

Figure9. The test automaton

by the *input_0* and *input_1* labels. After a request, say *input_1*, at the end of a bit cell($x = 18$), the sender changes the value of S_sig to 2, resets its clock and moves to the *mark_1* location. It remains there for the period of one clock cycle. At the end of that cycle the sender moves to to the location *code* and sets S_sig to the negation of its value at the end of the previous cell, which is then kept constant during the length of the mark subcell. At the beginning of the code subcell the value of S_sig is set again to 2 for the duration of one clock cycle and the sender moves to location *code2*. At the end of that cycle S_sig is set to the negation of its value at the end of the mark subcell and returns to the location *new_cell*. For each location v in this automaton, $div(v, x) = [1 - \epsilon, 1 + \epsilon]$, that is, the tolerance of the clock x is at most ϵ.

Receiver. The receiver is modeled by the receiver automaton in Fig. 11. See also Appendix A for the HyTech code. Initially the receiver is in the *edge_detect* location and the value of the receiver's clock y is equal to 1. In order to detect an edge in the signal train, the receiver uses the discrete variable R_prev with a domain $\{0, 1\}$, initially set to 1. R_prev represents the value of S_sig determined at the last sample. At the end of each clock cycle ($y = 1$) the receiver samples

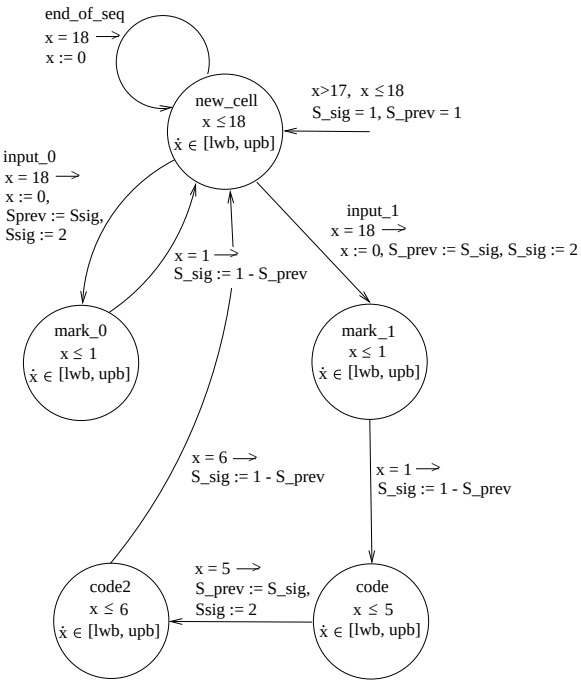

Figure10. The sender automaton

the value of S_sig. If the signal $s(t)$ has not achieved one of its extrema, that is, $S\_\text{sig} = 2$, the receiver must interpret this value nondeterministically as a 0 or as a 1. This is modeled by two actions: Namely, the action that moves to the *edge_detect* location, interpreting the value of S_sig as being equal to the value of S_sig at the time of the previous sample, hence no edge is detected; And the action that moves to the *receive* location, interpreting the value of S_sig as not being equal to the value of S_sig at the time of the previous sample, hence an edge is detected. If $S\_\text{sig} < 2$ the receiver just read the value of S_sig and depending on whether the value of R_prev is equal or not equal to the value of S_sig the automaton moves to the *edge_detect* location or it moves to the *receive* location. The receiver remains at the *receive* location for 10 clock cycles. At the end of the 10^{th} cycle it samples S_sig, passes the received bit to the test automaton, and returns to the *edge_detect* location. Again if $S\_\text{sig} = 2$, the receiver must interpret this value nondeterministically as a 0 or as a 1. For notational convenience, we have used the "not equal" symbol <>. In the HyTech code two cases are distinguished, since HyTech does not know disjunctions. For each location v in this automaton, $div(v, x) = [1 - \epsilon, 1 + \epsilon]$, that is, the tolerance of the clock x is at most ϵ. We choose the value of ϵ equal for both the sender–and the receiver–automata.

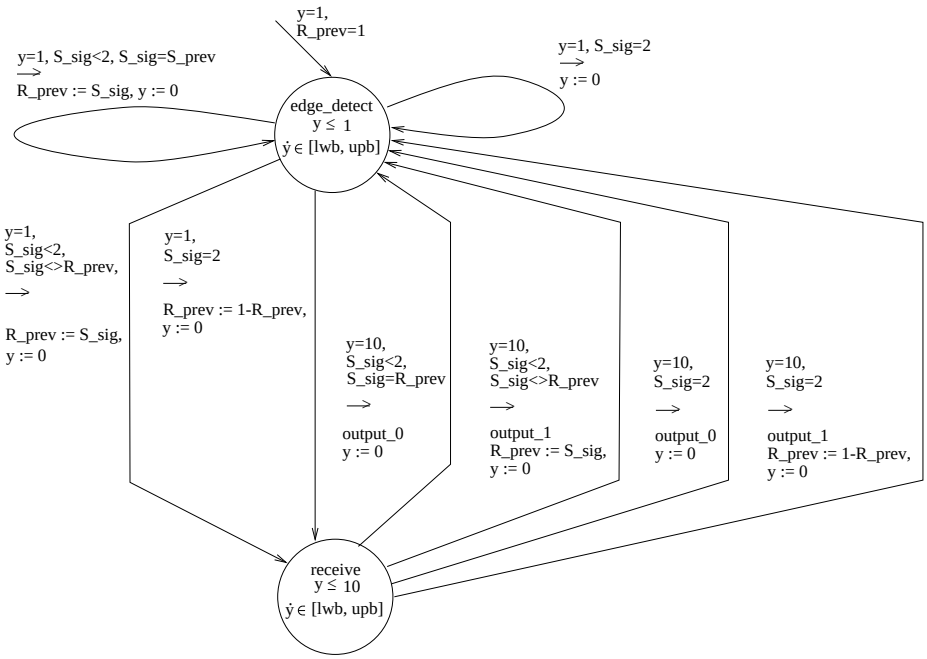

Figure11. The receiver automaton

4.2 Specification.

In this section we specify two requirements of our systems. We consider a safety requirement and a liveness requirement.

Safety Requirement. Our paradigm for specifying correctness is reachability analysis, in which we label certain states as violating. The system is correct if no violating states are reachable. In our case the violating states are specified as $l[test] = error$, that is, our system is not correct if the test automaton is in the *error* location. The following HyTech code checks the safety requirement, and generates an error trace if any exists, for error tolerance of 1/6.

```
var init_reg, final_reg, reached : region;
init_reg := loc[sender]=new_cell      & 17<x & x<=18 & S_sig=1
                                                    & S_prev=1
          & loc[receiver]=edge_detect & y=1         & R_prev=1
          & loc[test]=OK;
final_reg := loc[test]=error;
reached := reach forward from init_reg endreach;
if empty(reached & final_reg)
  then prints
```

```
       "Biphase Mark-18 verified for error tolerance 1/6";
  else prints
       "Biphase Mark-18 NOT verified for error tolerance 1/6";
       print trace to final_reg using reached;
endif;
```

Liveness Requirement. For verifying the liveness requirement we again use the paradigm of reachability analysis. We designed a test automaton which checks whether, from each reachable state in which the actions labeled with synchronization labels *input_1* or *input_0* are enabled, these actions indeed are taken. HyTech was not able to verify this. So, we design a simpler test environment. One automaton makes a request to the sender to transmit the bit sequence $(1, 0, 0, 1)$ and then stops. Another automaton checks if the bit sequence $(1, 0, 0, 1)$ is received. For all other received sequences the automaton moves to an error location. The sequence $(1, 0, 0, 1)$ is arbitrarily chosen. This simple test is a counter example to possibly trivially fulfilled safety requirement. See Appendix B for the HyTech code.

4.3 Analysis

The results of the analysis are summarized in Table 1.

Analysis of $bpm(18, 5, 10)$. Moore proves in [Moo94] the correctness of the configuration $bpm(18, 5, 10)$ for tolerance of $1/18$. This corresponds to a tolerance of 5 %. He also suggests that the clock rate restrictions can be considerably relaxed. His conjecture is that the protocol works for tolerance of almost 30 %, that is, $\epsilon \approx 43/143$.

We verify the safety requirement for $bpm(18, 5, 10)$ using an error tolerance ϵ of $1/5$ (20 %). HyTech successfully discovers that the violating states are not reachable. The correctness of the transmitted sequence is verified in 92.05 seconds[1]. See also Appendix D.

Next, we generated an error trace for $bpm(18, 5, 10)$ using an error tolerance of $1/4$ (see Appendix C). This means that the error tolerance for which $bpm(18, 5, 10)$ works is in the interval $[1/5, 1/4)$. We tried a tolerance of $9/40$, which lies in the middle of $[1/5, 1/4)$, but HyTech gave the error message: "Computation overflow. Error in Multiplication". This was also the reason why a parametric analysis did not succeed.

Analysis of $bpm(32, 16, 23)$. Moore proves in [Moo94] the correctness of $bpm(32, 16, 23)$ for tolerance of $1/32$ (3 %).

We verify the safety requirement for $bpm(32, 16, 23)$ using an error tolerance ϵ of $1/8$ (12.5%). HyTech successfully discovers that the violating states are not reachable. We also were able to generate an error trace for $bpm(32, 16, 23)$ using an error tolerance of $1/7$.

[1] All performance data in this paper were obtained on a SUN SPARCstation-20

Analysis of $bpm(16, 8, 11)$. The model of Moore [Moo94] failed to give a proof for the correctness of the configuration $bpm(16, 8, 11)$.

We verify the safety requirement for $bpm(16, 8, 11)$ using an error tolerance ϵ of 1/11 (9 %). HyTech successfully discovers that the violating states are not reachable. We also were able to generate an error trace for $bpm(16, 8, 11)$ using an error tolerance of 1/10.

Table1. Analysis results

Configuration	Moore's results	Results in this paper
$bpm(18, 5, 10)$	1/18 (correct) 43/143 (?)	1/5 (correct) 1/4 (incorrect) 9/40 (?)
$bpm(32, 16, 23)$	1/32 (correct)	1/8 (correct) 1/7 (incorrect)
$bpm(16, 8, 11)$		1/11 (correct) 1/10 (incorrect)

5 Conclusions

Linear hybrid automata enable a natural way of modeling the biphase mark protocol. Both discrete and continuous phenomena can be modeled directly in this formalism. The paradigm of reachability analysis enables us to specify safety- and liveness requirements. It was not obvious how to specify these requirements, but the literature gave us useful examples, which we used to model our requirements. Thanks to the very nature of linear hybrid automata we were able to relax one of the assumptions of Moore, namely that clock ticks are equally spaced events in real time. HyTech easily verified the correctness of biphase mark for wider clock drifts than those given in [Moo94]. We were not able to present a parametric analysis of the protocol because of restrictions on the size of the model that HyTech can currently handle.

A HyTech code for the Biphase Mark Protocol $bpm(18, 5, 10)$ (a safety requirement)

```
define(lwb,4/5)
define(upb,6/5)

var
        x,              -- sender's clock
        y,              -- receiver's clock
        : analog;
        S_sig,          -- value of the wire
        S_prev,         -- previous value of the wire
        R_prev,         -- previous value read by the receiver
        : discrete;
-- ------------------------------------------------------------
automaton sender
synclabs: input_1,      -- bit to send is 1
          input_0,      -- bit to send is 0
          end_of_seq;   -- no bit to send

initially new_cell & x>17 & x<=18 & S_sig=1 & S_prev=1;

loc new_cell: while x<=18 wait {dx in [lwb, upb]}
    when x=18 sync end_of_seq do {x'=0} goto new_cell;
    when x=18 sync input_0 do {x'=0, S_prev'=S_sig, S_sig'=2}
                                                  goto mark_0;
    when x=18 sync input_1 do {x'=0, S_prev'=S_sig, S_sig'=2}
                                                  goto mark_1;

loc mark_0: while x<=1 wait {dx in [lwb, upb]}
    when x=1 do {S_sig'=1-S_prev} goto new_cell;

loc mark_1: while x<=1 wait {dx in [lwb, upb]}
    when x=1 do {S_sig'=1-S_prev} goto code;

loc code: while x<=5 wait {dx in [lwb, upb]}
    when x=5 do {S_prev'=S_sig, S_sig'=2} goto code2;

loc code2: while x<=6 wait {dx in [lwb, upb]}
    when x=6 do {S_sig'=1-S_prev} goto new_cell;
end -- sender
-- ------------------------------------------------------------
automaton receiver
synclabs : output_1,    -- received bit is 1
           output_0;    -- received bit is 0
```

```
initially edge_detect & y=1 & R_prev=1;

loc edge_detect: while y<=1 wait {dy in [lwb, upb]}
    when y=1 & S_sig<2 & S_sig=R_prev do {R_prev'=S_sig, y'=0}
                                            goto edge_detect;
    when y=1 & S_sig<2 & S_sig>R_prev do {R_prev'=S_sig, y'=0}
                                                goto receive;
    when y=1 & S_sig<2 & S_sig<R_prev do {R_prev'=S_sig, y'=0}
                                                goto receive;
    when y=1 & S_sig=2             do {y'=0}
                                            goto edge_detect;
    when y=1 & S_sig=2             do {R_prev'=1-R_prev, y'=0}
                                                goto receive;

loc receive: while y<=11 wait {dy in [lwb, upb]}
    when y=11 & R_prev=S_sig sync output_0
                 do {y'=0, R_prev'=S_sig} goto edge_detect;
    when y=11 & R_prev=1 & S_sig=0 sync output_1
                 do {y'=0, R_prev'=0} goto edge_detect;
    when y=11 & R_prev=0 & S_sig=1 sync output_1
                 do {y'=0, R_prev'=1} goto edge_detect;
    when y=11 & S_sig=2 sync output_0
                 do {y'=0} goto edge_detect;
    when y=11 & S_sig=2 sync output_1
                 do {y'=0, R_prev'=1-R_prev} goto edge_detect;
end -- receiver
-- ------------------------------------------------------------
automaton test
synclabs: output_1,    -- received bit is 1
          output_0,    -- received bit is 0
          input_1,     -- bit to send is 1
          input_0,     -- bit to send is 0
          end_of_seq;  -- no bit to send

initially OK;

loc OK: while True wait {}
    when True sync end_of_seq goto OK;
    when True sync input_1    goto I1;
    when True sync input_0    goto I0;
    when True sync output_0   goto error;
    when True sync output_1   goto error;


loc I0: while True wait {}
```

```
    when True sync output_0   goto OK;
    when True sync output_1   goto error;
    when True sync end_of_seq goto error;
    when True sync input_1    goto error;
    when True sync input_0    goto error;

loc I1: while True wait {}
    when True sync output_0   goto error;
    when True sync output_1   goto OK;
    when True sync end_of_seq goto error;
    when True sync input_1    goto error;
    when True sync input_0    goto error;

loc error: while True wait {}
end -- test
-- ------------------------------------------------------------
-- analysis commands
var
init_reg, final_reg, reached : region;

init_reg := loc[sender]=new_cell & 17<x & x<=18 & S_sig=1
                                  & S_prev=1
          & loc[receiver]=edge_detect & y=1 & R_prev=1
          & loc[test]=OK;

final_reg := loc[test]=error;

reached := reach forward from init_reg endreach;

if empty(reached & final_reg)
  then prints
       "Biphase Mark-18 verified for error tolerance 1/6";
  else prints
       "Biphase Mark-18 NOT verified for error tolerance 1/6";
       print trace to final_reg using reached;
endif;
-- analysis commands
```

B HyTech code for the Biphase Mark Protocol $bpm(18, 5, 10)$ (a liveness requirement)

```
automaton testInput
synclabs: input_1,     -- bit to send is 1
          input_0;     -- bit to send is 0

initially OK;

loc OK: while True wait {}
    when True sync input_1  goto I1;


loc I1: while True wait {}
    when True sync input_0  goto I10;

loc I10: while True wait {}
    when True sync input_0  goto I100;

loc I100: while True wait {}
    when True sync input_1  goto stop;

loc stop: while True wait {}
end -- testInput

automaton testOutput
synclabs: output_1,     -- bit to send is 1
          output_0;     -- bit to send is 0

initially OK;

loc OK: while True wait {}
    when True sync output_1  goto I1;
    when True sync output_0  goto error;

loc I1: while True wait {}
    when True sync output_0  goto I10;
    when True sync output_1  goto error;

loc I10: while True wait {}
    when True sync output_0  goto I100;
    when True sync output_1  goto error;

loc I100: while True wait {}
    when True sync output_1  goto stop;
    when True sync output_0  goto error;
```

```
loc stop: while True wait {}
    when True sync output_0  goto error;
    when True sync output_1  goto error;

loc error: while True wait {}
end -- testInput

-- analysis commands
var
init_reg, final_reg1, final_reg2, reached : region;

init_reg := loc[sender]=new_cell     & 15<x & x<=16 & S_sig=1
                                                    & S_prev=1
          & loc[receiver]=edge_detect & y=1         & R_prev=1
          & loc[testInput]=OK
          & loc[testOutput]=OK;

final_reg1 := loc[testOutput]=error;
final_reg2 := loc[testOutput]=stop;

reached := reach forward from init_reg endreach;

if empty(reached & final_reg1)
  then prints "Location ‘‘error’’ is NOT reachable";
  else prints "Location ‘‘error’’ is reachable";
       print trace to final_reg1 using reached;
endif;

if empty(reached & final_reg2)
  then prints "Message ‘‘1001’’ is NOT received";
  else prints "Message ‘‘1001’’ is received";
       print trace to final_reg2 using reached;
endif;
```

C Trace to the *error* location of $bpm(18, 5, 10)$ for tolerance of 1/4

```
Command: /home/infstud/sivanov/HyTech/bin/hytech -o2 bfm18
================================================================
HyTech: symbolic model checker for embedded systems
Version 1.04 10/15/96
For more info:
    email: hytech@eecs.berkeley.edu
    http://www.eecs.berkeley.edu/~tah/HyTech
Warning: Input has changed from version 1.00(a).
Use -i for more info
================================================================
Will try hard to avoid library arithmetic overflow errors
Number of iterations required for reachability: 20
Biphase Mark-18 NOT verified for error tolerance 1/4
 ====== Generating trace to specified target region ========
Time: 0.00
Location: new_cell.edge_detect.OK
     x = 18 & y = 1 & S_sig = 1 & S_prev = 1 & R_prev = 1
------------------------------
  VIA: input_1
------------------------------
Time: 0.00
Location: mark_1.edge_detect.I1
    x = 0 & y = 1 & S_sig = 2 & S_prev = 1 & R_prev = 1
------------------------------
  VIA:
------------------------------
Time: 0.00
Location: mark_1.receive.I1
     x = 0 & y = 0 & S_sig = 2 & S_prev = 1 & R_prev = 0
---------------
 VIA 1.33 time units
---------------
Time: 1.33
Location: mark_1.receive.I1
     x = 1 & 3y = 5 & S_sig = 2 & S_prev = 1 & R_prev = 0
------------------------------
  VIA:
------------------------------
Time: 1.33
Location: code.receive.I1
     x = 1 & 3y = 5 & S_sig = 0 & S_prev = 1 & R_prev = 0
---------------
 VIA 5.33 time units
```

```
---------------
Time: 6.67
Location: code.receive.I1
     x = 5 & 3y = 25 & S_sig = 0 & S_prev = 1 & R_prev = 0
-------------------------------
  VIA:
-------------------------------
Time: 6.67
Location: code2.receive.I1
     x = 5 & 3y = 25 & S_sig = 2 & S_prev = 0 & R_prev = 0
---------------
 VIA 1.33 time units
---------------
Time: 8.00
Location: code2.receive.I1
     x = 6 & y = 10 & S_sig = 2 & S_prev = 0 & R_prev = 0
-------------------------------
  VIA: output_0
-------------------------------
Time: 8.00
Location: code2.edge_detect.error
     x = 6 & y = 0 & S_sig = 2 & S_prev = 0 & R_prev = 0
 ============ End of trace generation ============


================================================================
Max memory used =   666 pages =  2727936 bytes =   2.60 MB
Time spent      =     119.26u +      1.75s =     121.01 sec
                                                       total
================================================================
```

D Computational data for the verification of $bpm(18, 5, 10)$

```
Command: /home/infstud/sivanov/HyTech/bin/hytech -o2 bfm18
============================================================
HyTech: symbolic model checker for embedded systems
Version 1.04 10/15/96
For more info:
    email: hytech@eecs.berkeley.edu
    http://www.eecs.berkeley.edu/~tah/HyTech
Warning: Input has changed from version 1.00(a).
Use -i for more info
============================================================

Will try hard to avoid library arithmetic overflow errors
Number of iterations required for reachability: 22
Biphase Mark-18 verified for error tolerance 1/5
============================================================
Max memory used =   611 pages =  2502656 bytes =   2.39 MB
Time spent      =      90.88u +      1.17s =      92.05 sec
                                                      total
============================================================
```

References

[ACHH93] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In Grossman et al. [GNRR93], pages 209–229.

[AHH93] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the 14th Annual Real-time Systems Symposium*, pages 2–11. IEEE Computer Society Press, 1993.

[BG87] D. P. Bertsekas and R. G. Gallager. *Data Networks*. Prentice Hall, 1987.

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

[Cor91] Intel Corporation. *Microcommunications*. Intel Literature Sales, P. O. Box 7641, Mt. Prospects, IL, 1991.

[GNRR93] R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors. *Hybrid Systems*, Lecture Notes in Computer Science 736. Springer-Verlag, 1993.

[Hen92] T.A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.

[HHWT95] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HyTech. In E. Brinksma, W.R. Cleaveland, K.G. Larsen, T. Margaria, and B. Steffen, editors, *TACAS 95: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1019, pages 41–71. Springer-Verlag, 1995.

[HWT95] P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In *Proceedings of the Seventh International Conference on Computer-aided Verification*, Lecture Notes in Computer Science 939, pages 381–395. Springer-Verlag, 1995.

[HWT96] T.A. Henzinger and H. Wong-Toi. Using HyTech to synthesize control parameters for a steam boiler. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, Lecture Notes in Computer Science 1165, pages 265–282. Springer-Verlag, 1996.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

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

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