Machine-Assisted Parameter Synthesis of the Biphase Mark Protocol Using Event Order Abstraction*

Shinya Umeno

CSAIL, Massachusetts Institute of Technology, Cambridge MA, USA
umeno@csail.mit.edu

Abstract. We present machine-assisted timing-parameter synthesis of the biphase mark protocol (BMP) [1] using *event order abstraction* (EOA)[2]. By using EOA, we separate the task of synthesizing parameter constraints that guarantee key safety properties of BMP into two parts: 1. Safety property verification of the protocol by a conventional untimed model-checker under the condition that “bad” event orders do not occur; and 2. Derivation of timing parameter constraints that are sufficient to exclude bad event orders in the protocol, using our tool METEORS. Though the user has to provide information about bad event orders, the rest of the synthesis process is automated. With the case study presented in this paper, we provide the community with two new pieces of information about BMP. First, the synthesis process using EOA produces, as a by-product, a list of all “bad scenarios” of BMP that would happen when parameters are tuned incorrectly. Second, the METEORS tool provides information about which parameter constraint in the finally derived conjunction of constraints is actually sufficient to exclude each of these bad scenarios.

1 Introduction

Time-parametric verification of real-time systems has been a challenging problem in the formal verification community [1, 3, 4]. In this problem, timing constraints on the systems’ behavior depend on a certain parameter set, and values of these parameters are not fixed at the time of verification. Typically, only a subset of possible parameter combinations in the entire parameter space satisfies correctness of such systems. Thus, to conduct parametric verification of them, the user has to find an appropriate set of constraints for the timing parameters, and manually proves or mechanically checks correctness under the constraints.

Timing-parameter synthesis is a problem in which one wants to derive with a machine support sufficient constraints on the timing parameters of an underlying real-time system under which the system executes correctly. This problem is considered harder than time-parametric verification since the mechanical tool is not a priori given a set of timing parameter constraints by the user, but it has to synthesize constraints by itself.

In this paper, we present machine-assisted timing-parameter synthesis of the biphase mark protocol (BMP) [1] using *event order abstraction* (EOA) that we presented in [2].[1]

* This work is supported by the NSF Award CCF-0702670 and the NSF Award CNS-0614414. This paper is formatted using the LNCS LATEX template. The paper will appear in The 7th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 2009), Budapest, Hungry, September 13-16, 2009.

[1] The SAL code and the python code to reproduce the results of the presented case study are available from: `http://people.csail.mit.edu/umeno/biphase/`.

EOA can be applied to real-time systems in which "correct orderings of events" maintained by timing constraints on the systems' behavior are critical for correctness. To use EOA, the user models a real-time system by using the *time-interval automata* (TIA) framework, an extension of the I/O automata framework [5]. By using EOA, we separate the task of synthesizing parameter constraints that guarantee key safety properties of BMP into two parts: 1. Safety property verification of the protocol by a conventional untimed model-checker under the condition that "bad" event orders do not occur; and 2. Derivation of timing parameter constraints that are sufficient to exclude bad event orders in the protocol, using our tool METEORS. Though the user has to provide information about bad event orders, the rest of the synthesis process is completely automated.

BMP has been studied in several papers in the context of time-parametric verification, which requires a manual derivation of parameter constraints (for instance, [1, 3, 6]), and in the context of fully automated constraint synthesis for a restrictedly parameterized model of the protocol (in which some parameters are fixed to constraint values) [7, 8]. This work presents the first machine-assisted parameter synthesis for fully parameterized model of BMP. This protocol has interesting aspects for timing parameter synthesis: 1. Considerably large number of timing parameters (10 parameters in our model) and 2. Unbounded number of repetitive events (busy-waiting) with timing constraints that may appear in the protocol executions. We will explain in Section 2 why the above mentioned two aspects make this protocol especially challenging to conduct timing-parameter synthesis with existing model-checkers. We also compare the results with case studies on BMP using other approaches.

With this case study presented in this paper, we provide the community with two new pieces of information about BMP. First, the synthesis process using EOA produces, as a by-product, a list of all "bad scenarios" of BMP that would happen when parameters are tuned incorrectly. Second, the METEORS tool provides information about which parameter constraint in the finally derived conjunction of constraints is actually sufficient to exclude each of these bad scenarios. We believe that these pieces of information by themselves deserves values to the community. For example, by the list of all bad scenarios and its corresponding parameter constraint, an implementer of BMP can learn what kind of bad executions could occur in the protocol when the parameters are badly tuned or the values of parameters temporally deviate from their nominal values due to transient failures. When a bad execution is actually observed in an implementation, by the list of parameter constraints for each bad scenarios, the implementer can discover which parameter constraint is violated. The user cannot easily obtain these pieces of information about bad executions of the system by existing model-checkers.

The rest of the paper is organized as follows. In Section 2, we explain the reason why BMP is especially challenging to conduct fully automatic parameter synthesis with existing model-checkers. Section 3 is devoted to summarize the event order abstraction (EOA) approach that we use for the case study. In Section 4, we conduct machine-assisted parameter synthesis of BMP using EOA. We also compare the results with existing case studies on BMP using other approaches. In Section 5, we conclude by discussing advantages of using EOA and stating future work.

2 Why Is the Biphase Mark Protocol Especially Challenging?

There are several existing timed/hybrid model-checkers that can conduct fully automatic timing-parameter constraint synthesis [7, 9–12]. However, the biphase mark pro-

tocol (BMP) is especially challenging to conduct parameter constraint synthesis by these model-checkers from two reasons that we will describe in the following.

The first reason is that BMP performs repetitions of events, and there is a time bound between two consecutive repetitions. Namely, the receiver part of BMP conducts busy waiting to detect a signal edge, and checks to detect an edge are performed repeatedly in every certain time interval. As we will see in Section 4, these repeated checks could fail an unbounded times since we do not know the relationship between the repetition cycle (parameter Δ) and other timing parameters that are related to the timing of creating an edge. The second reason is that BMP has a considerable number of timing parameters (10 parameters in our modeling). Thus even if the user eliminates repetitions of events with timing constraints by modifying the model, the resulting model may have too many parameters to fully automatically synthesis parameter constraints. Indeed, in [7], the authors had to fix some parameters in order to automatically synthesize parameter constraints by HYTECH.

Now let us explain why the first reason causes a trouble for existing model-checkers. The basics of an existing timed/hybrid model-checker performs parameter constraint synthesis are as follows. It first computes the reachable states of the system, symbolically represented by a linear-arithmetic expression. This is done by repeatedly computing successor states until no new successor states are discovered. Then, by taking the intersection of the reachable state and the unsafe states, it obtains the bad parameter settings. Now a parameter constraint is obtained by negating the linear-arithmetic expression that represents the intersection. If no over- or under-approximation is used for reachable set computation, then the constraint is both sufficient and necessary. When the user gives a fully parameterized model of BMP to a model-checker, the reachability computation does not terminate. As we explained above, arbitrarily number of failing checks can potentially occur by busy waiting in BMP. Because these detections are repeatedly performed every Δ time units, every successor state of these repeated detections is a new state in the reachability computation (for example, the state that can be reached just after two failing detections is a different state from the state just after one failing detection since Δ time units are elapsed since the first detection). Therefore, the reachability computation diverges. Indeed, in [7], Henzinger et al. explained that this was the reason that they had to modify the HyTech model of BMP in such a way that the model does not perform busy-waiting for an edge detection: a successful detection occurs within a certain time after an edge has created (and thus they eliminated failing detections from the model).

Our EOA approach does not suffer from the same problem since the automaton model is untimed in our approach. In this untimed abstraction, failing detections become stuttering transitions, and thus the model-checking does not diverge due to failing detections. Instead, the constraint derivation process from bad event orders of EOA needs a technique called "decomposition" of event orders. This process is automated by the METEORS tool once the user specifies what events must be decomposed. We will present more details about decomposition in Section 3.3.

The TReX [11] model-checker can perform what the authors call "extrapolation", with which the tool detects the loop in the state transition graph and over-approximates the effects of the loop in terms of changes of the values of timed variables. For example, for the above discussed repetitions of failing detections, TReX may be able to compute with the extrapolation technique that the effect of these repetitions is increasing the values of timed variables by $n\Delta$ for an arbitrarily non-negative number n, and

therefore may be able to complete the reachable state computation. However, as far as we can observe from [13], which presents timing-parameter synthesis for IEEE 1394 root contention protocol using TReX, the tool actually needs human-directed operations for choosing parameter constraints: the tool gives back the user the list of linear inequalities which sometimes contains unnecessary inequalities for the correctness (due to over-approximation used in the model-checking), and the user needs to manually choose a subset of the given set of inequalities that he/she believes sufficient and repeat the model-checking under the selected constraints. Moreover, the authors reported that the computation took more than 67 hours for a certain model-checking run under selected constraints. Considering that their model only have five parameters, which are considerably less compared to ten parameters in our model of BMP, timing synthesis of BMP using TReX is arguably very challenging.

3 Event Order Abstraction Approach

In this section, we explain the event order abstraction (EOA) approach. In Section 3.1, we describe the time-interval automata framework that the user needs to use to model a system when using EOA. Section 3.2 is devoted to explain how the user can specify bad event orders in the system. Finally in Section 3.3, we summarize how the user conducts timing parameter synthesis using EOA and our timing-constraint synthesis tool METEORS. For more detailed explanation of EOA, the reader should refer to [2].

3.1 Time-Interval Automata

The *time-interval automata* (TIA) framework is an extension of the I/O automata framework [5]. An I/O automaton is a classical transition system with distinguished input, output, and internal actions, and is usually described by a guarded-command style language. Informally, with the TIA framework, one can specify the lower and upper time bounds on the interval between one action and its following actions. A time bound for an action a and actions in a set of actions B is represented as an interval in the form $[l, u]$. Informally, this bound represents that, for any time of occurrence t_a of action a, no action in B occurs before $t_a + l$, and at least one action in B is performed before or at $t_a + u$.

An *interval-bound map* defined in the following Definition 1 formally specifies time bounds for actions. The special symbol $\perp$ is used to express the time bound on the interval between the system start time and the time an action in the specified set occurs.

Definition 1. *(Interval-bound map). An interval-bound map b for an I/O automaton A is a pair of mappings, $lower$ and $upper$. Each of $lower$ and $upper$ is a partial function from $actions(A)_\perp \times \mathcal{P}(actions(A))$ to $\mathrm{R}^{>0}$, where $actions(A)_\perp = actions(A) \cup \{\perp\}$ is a set of actions of A extended with a special symbol $\perp$, $\mathcal{P}(actions(A))$ is the power set of actions of A, and $\mathrm{R}^{>0}$ is the set of positive reals.*

An interval-bound map defined in Definition 1 may not satisfy requirements to express a meaningful bound (for example, the specified lower bound is not greater than the specified upper bound). The formal description of the requirements appears in [2]. We say that an interval-bound map is *valid* if it satisfies the requirements.

Definition 2. *(Time-interval automaton). A time-interval automaton (A, b) is an I/O automaton A together with a valid interval-bound map b for A.*

Definition 3. *(Timed execution). A timed execution of a time-interval automaton* (A, b) *is a (possibly infinite) sequence* $\alpha = s_0, (\pi_1, t_1), s_1, (\pi_2, t_2), \cdots$ *where the* s_i*'s are states of* A*, the* π_i*'s are actions of* A*, and the* t_i*'s are times in* $R^{\geq 0}$*;* s_0 *is an initial state of* A*; and for any* $j \geq 1$*,* (s_{j-1}, π_j, s_j) *is a valid transition of* A *and* $t_j \leq t_{j+1}$*. We also require a timed execution to satisfy the upper and lower bound requirements expressed by* b*:*
Upper bound: *For every pair of an action* π *and a set of actions* Π *with* $upper(\pi, \Pi)$ *defined, and every occurrence of* π *in the execution* $\pi_r = \pi$*, if there exists* $k > r$ *with* $t_k > t_r + upper(\pi, \Pi)$*, then there exists* $k' > r$ *with* $t_{k'} \leq t_r + upper(\pi, \Pi)$ *and* $\pi_{k'} \in \Pi$*.*
Lower bound: *For every pair of an action* π *and a set of actions* Π *with* $lower(\pi, \Pi)$ *defined, and every occurrence of* π *in the execution* $\pi_r = \pi$*, there does not exist* $k > r$ *with* $t_k < t_r + lower(\pi, \Pi)$ *and* $\pi_k \in \Pi$*.*
The upper and lower bound requirements for a bound with $\perp$ *are defined similarly (see [2]).*

A composition of multiple TIA is defined in a way similar to that of ordinary I/O automata (which is an ordinary asynchronous composition with synchronization of input and output actions with the same name [5]). Interval-bound maps of TIA are combined by using a union of maps (by regarding maps as relations). In order to formally define a composition for time-interval automata, we need a definition of the compatibility of a collection of TIA. The compatibility for TIA is defined simply as the compatibility of the underlying I/O automata (see [5] for the definition).

Definition 4. *(Composition of TIA) For a compatible collection of TIA, the composition* $(A, b) = \Pi_{i \in I}(A_i, b_i)$ *is the timed-interval automaton as follows. (1).* A *is the composition of the underlying I/O automata* $\{A_i\}_{i \in I}$*, and (2).* $lower$ *is given by taking union of* $\{lower_i\}_{i \in I}$ *and* $upper$ *is given by taking union of* $\{upper_i\}_{i \in I}$ *(by regarding partial functions as sets of ordered pairs).*
Definition 5. *(Untimed TIA) Given a TIA* (A, b)*, the untimed model of* (A, b) *is simply an underlying ordinary untimed I/O automaton* A*.*

3.2 Specifying Event Orders

In this section, we presents how the user can specify an event order that needs to be excluded for system correctness. One event order specification represents a subsequence of an (untimed) execution (or technically called *execution fragment*) or a set of execution fragments.

An event order in its simplest form is just a sequence of actions (transition labels), which represents consecutive actions that occur in an automaton execution. In some cases (such as certain bad scenarios of BMP), it is crucial to express repetitions of events. The user can start an event order specification with the special symbol '$\perp$', which indicates that the event order matches a prefix of an automaton execution, rather than an execution fragment in the middle of the execution.

The user can express repetitions using an *ignored event specification* (IES). An IES specifies the repetitive events in a way similar to the repetition symbol '*' of regular expressions. For example, an event order with an ignored event specification "a_1-a_2-a_3-a_4: insert $\{a_5, a_6\}$ in [2,4]" matches with any execution of TIA that has a subsequence that matches a regular expression $a_1 a_2 (a_5 \cup a_6)^* a_3 (a_5 \cup a_6)^* a_4$. We use the above notation using "insert", instead of a repetition symbol '*', since we consider that it is easier to comprehend which events are inserted in what event interval.

Definition 6. *(Event order) An event order of a time-interval automaton (A, b) is a sequence of actions of A, possibly starting with a special symbol $\perp$.*

In Definition 7, Y_m represents a set of events that are inserted in the interval between e_{i_m} and e_{j_m}.

Definition 7. *(Ignored event specification). An ignored event specification (IES) for an event order is in the following form:* $\mathsf{insert}\ (Y_m\ \mathsf{to}\ [i_m, j_m])_{m=1}^{r}$.

An ignored event set I_k^E represents the set of all ignored events between event index k and $k+1$.

Definition 8. *(Ignored event set). For an event order with an IES, $E = (\perp)e_1 \cdots e_n$:* $\mathsf{insert}\ (Y_m\ \mathsf{to}\ [i_m, j_m])_{m=1}^{r}$*, we define $I_k^E = \bigcup_{i_m \leq k < j_m} Y_m$ for $0 \leq k \leq n-1$.*

Definition 9. *(Match between a timed execution and an event order with an IES). Consider a timed execution $\alpha = s_0, (\pi_1, t_1), s_1, \cdots$ of a time-interval automaton (A, b). Let α' be the sequence of actions that appear in α, that is, $\alpha' = \pi_1 \pi_2 \pi_3 \cdots$. We say that α matches an event order (with an IES), $E = e_1 \cdots e_n$:* $\mathsf{insert}\ (Y_m\ \mathsf{to}\ [i_m, j_m])_{m=1}^{r}$*, if there exists a finite subsequence β of α' such that β can be split into $\beta_0 \pi_{k_1} \beta_1 \pi_{k_2} \beta_2 \cdots \beta_{n-1} \pi_{k_n}$, where, for all i, $1 \leq i \leq n$, $\pi_{k_i} = e_i$, and β_i is a sequence of actions and all actions that appear in β_i are in I_i^E.*

A match for an event order that starts with $\perp$ is defined similarly to Definition 9 (an additional condition $k_1 = 1$ is added to the definition). For an event order without an IES, all β_i's in Definition 9 are empty sequences.

3.3 Timing-Parameter Synthesis using EOA

Timing parameter synthesis using EOA involves two steps: 1. Identifying bad event orders in the untimed abstraction of the original model; and 2. Deriving timing parameter constraints using our METEORS tool. We explain these two steps in more details in the following.

The first step is identification of "bad" event orders that appear in the system executions. Note that the user does not need to specify all bad executions, but it is sufficient to specify the set of key subsequences of them that covers all bad executions.

The user initially proposes a candidate set of bad event orders that he/she wants to exclude from the system executions (this candidate set may be empty). The user then model-checks a safety property of interest on an *untimed model* of the underlying TIA, under the assumption that the model does not exhibit the proposed bad event orders. Recall that an untimed model of a TIA (A, b) is simply an underlying ordinary I/O automaton A. If the model-checking is completed with a positive answer, then the user has obtained a sufficient set of bad event orders to be excluded for the given safety property. Otherwise, the user uses a counterexample obtained from the model-checking to extract an additional bad event order, and repeats the same process until he/she successfully model-checks the untimed model.

Model-checking under a specific event order assumption is carried out in the following two steps. The user first constructs a monitor that raises a flag when any of the identified bad event orders is exhibited. Then he/she model-checks the untimed model with this monitor under the assumption that the monitor does not raise the flag (in Linear Temporal Logic (LTL) [14], this condition can be represented by: $\Box(\neg\mathsf{Monitor.flag}) \Rightarrow \Box(\neg\mathsf{UntimedModel.propertyViolated})$). We used the SAL model-cheker [15] in the presented work. We manually constructed monitors in the presented case study, but we are planning to develop an automatic monitor construction tool.

In the second step, the user provides the set of identified bad event orders to our tool METEORS (MEchanized Timing/Event-ORder Synthesizer), and the tool automatically derives timing parameter constraints under which the underlying TIA exhibits no execution that matches the identified bad event orders. The algorithm that METEORS uses and a soundness theorem of the derived constraints are described in [2]. We briefly summarize the basic idea of the algorithm here. Given an event order, the tool combines the time bounds immediately derivable from the interval-bound map b of the underlying TIA (A, b), and finds a pair of a combined upper bound and a combined lower bound. For example, suppose the tool is given the event order $E = e_1 e_2 e_3 e_4$, and is able to derive from b upper bounds U_1 for $[e_1, e_3]$ and U_2 for $[e_2, e_4]$, respectively, and a lower bound ℓ for $[e_2, e_4]$. (We use the notation $[e_i, e_j]$ to represent the time interval between events e_i and e_j in E.) If $U_1 + U_2 < \ell$ holds, then E cannot be exhibited by (A, b) since the time interval between e_1 and e_4 in E is at most $U_1 + U_2$ and the time interval between e_2 and e_4 is at least ℓ, and thus e_3 must appear before e_4 does after $e_1 e_2$. The algorithm that METEORS uses systematically goes through all possible combinations of upper and lower bounds to find a constraint using the same reasoning described above. An IES is treated very conservatively: we basically literally ignore ignored events (except for some subtle cases). This is basically why we need *decomposition* of an event order with an IES, described below, for some cases to retrieve some more information of repetitive events (than just ignoring them), in order to derive more meaningful constraints.

We have added a new feature of automatic decomposition of event orders to METEORS after we presented [2]. With this new feature, the user can specify repetitive events specified in an IES that he/she wants to "decompose". A decomposition of events in an IES is sometimes needed to obtain a weaker constraint set for the correctness (representing a larger allowable parameter set) than the one that could have been obtained without decomposition. A decomposition of an event order with an IES creates two event orders such that a union of the two sets of executions that decomposed two orders match respectively is equal to the set of executions that the original event order matches. This is done by splitting the event order into the case that specified repetitive events occur at least one time and the case that no repetition occurs. For example, decomposition of event order $E_1 = a_1(a_2)^* a_3 a_4$ with respect to $(a_2)^*$ produces two event orders $E_1^0 = a_1 a_3 a_4$ and $E_1^+ = a_1(a_2)^* a_2 a_3 a_4$, where $E_1^0 \cup E_1^+ = E_1$. Now the constraint for the underlying event order is derived not directly from the original one, but from the two decomposed ones. The user can basically command the tool to decompose all repetitions of events (for example failing detections in BMP), and the tool will automatically decompose them.

A constraint derived from one event order by METEORS has the form of a disjunction of linear inequalities over the upper and lower bound parameters (one event order may have several parameter inequalities for it to be excluded). The user typically has to exclude more than one event orders, and thus the tool needs to combine individual constraints by making a conjunction of the constraints. Thus, a parameter constraint derived from METEORS forms a *conjunction of disjunctions of linear inequalities*.

By combining the untimed model-checking result from the first step and the derived constraint from the second step, the user obtains a timing parameter constraint under which the underlying TIA model satisfies the desirable safety property.

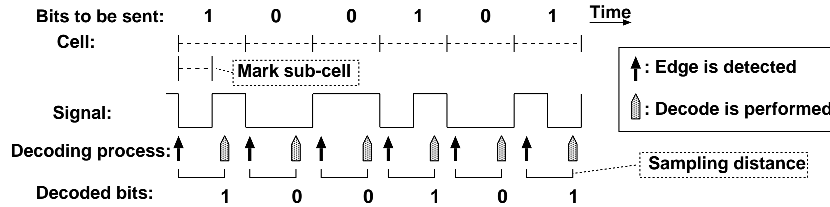

Fig. 1. Basic execution of the biphase mark protocol

4 Case Study: the Biphase Mark Protocol

The biphase mark protocol (BMP) is a widely used lower-layer communication protocol for industrial and consumer electronics. For example, it is used in Sony/Philips Digital Interconnect Format (S/PDIF) that has been developed for carrying digital audio signals between devices and stereo components. An industrial version of S/PDIF, called AES/EBU, also uses BMP.

BMP specifies a way of encoding a bit string to a digital signal, and then decoding the signal back to a bit string using the timing aspect of the encoded signal. Figure 1 shows how BMP operates. The encoder and the decoder communicate via a digital signal (a function from time to 'high' and 'low') sent on a physical wire.

At the encoder side, the time frame is divided into small time windows, called *cells*. In each cell, the encoder encodes one bit into a digital signal at a time. The encoding rule specified in BMP is simple. At the beginning of every cell, the encoder flips the signal, creating an *edge*. Within one cell, when the encoder has to encode a '1', it flips the signal within the cell after some time window called *mark sub-cell*. On the other hand, when encoding a '0', the encoder simply does not flip the signal withing the cell.

The decoder repeatedly checks the signal, and detects the edge by observing that the level of the signal (low or high) is different from what it observed in the last check. The decoder interprets the detected edge as the beginning of a cell, and checks the signal again after the time length called the *sampling distance*. If the decoder observes the level of the signal changes within the cell, it decodes a '1', and otherwise a '0'.

Several researchers have conducted formal verification of this protocol (for example, [1, 3, 6]). We compare the results from the EOA approach with related work in Section 4.3. The TIA model of a biphase mark protocol we have developed is based on the model by Vaandrager and de Groot [3]. In the model presented in [3], the authors consider two important realistic aspects of the protocol. The first aspect is differences in the clock rates of the encoder and the decoder (that is, *clock drift*). From this aspect, the sizes of cells (and mark sub-cells) are not uniformly consistent in the protocol execution, but have some small deviation from its ideal shape (that is, for an ideal cell size L, an actual cell has a size within $[L - \varepsilon, L + \varepsilon]$). The same phenomenon affects the decoder as well since its periodic checks for a signal edge detection and the sampling period depend on its local clock. The second aspect is the *metastability* of the signal caused by a signal edge. When a signal edge is created by the encoder, the signal level does not immediately change from a high voltage to low, or low to high, but needs some settling time. When the decoder checks the signal during this metastability period, we cannot predict whether the decoder may interpret the signal level as high or low. Thus, as in the model of [3], the observation of the signal by the encoder is nondeterministically decided as either high or low within a metastability period, so that we cover all possible scenarios that arise from this metastability issue.

4.1 Modeling BMP in the TIA Framework

Figure 2 (Encoder automaton) shows TIA code for the model of the encoder in BMP. This automaton performs the following simple job: it repeatedly and nondeterministically chooses the next bit to send (expressed by choose(sending0,sending1S)). It first outputs Edge0 or Edge1S, depending on the next bit to send. If it chooses to send a '1', then it outputs Edge1T, representing the "toggling" of the signal within the current cell, before choosing the next bit to send. The automaton has three bounds. The first bound represents that the size of mark sub-cell (the time it waits between the edge starting a cell and the toggling of the signal to encode a '1') is within $[m_1, M_1]$. The second and the third bound represent that the entire cell length is within $[c, C]$.

Figure 4 (Decoder automaton) shows TIA code for the model of the decoder in BMP. This automaton also models signal settling (Settle action) on the wire as well as the behavior of the decoder (Detect and Decode actions). The code is a straightforward translation of the decoder behavior explained earlier in this Section 4. We use the temporary variable sampledVoltage to represent a sampled voltage of the signal on the wire at the decoder side. When the signal has not yet settled after an edge is created by the encoder, the value of sampledVoltage is nondeterministically determined (expressed by choose(HIGH,LOW)). Detect(true) and Detect(false) respectively represents that the decoder succeeds in detecting, and fails to detect a signal-level change.

Decoder has seven time-interval bounds.[2] The first bound specifies that the very first detection of an edge is performed within the time interval $[\delta, \varDelta]$ after the system execution starts. The second bound specifies that after the failing detection of an edge, the next detection is performed within $[\delta, \varDelta]$. The third bound specifies that the decoder resumes an edge detection within $[\delta, \varDelta]$ after decoding a bit. The fourth bound specifies that the sampling distance is within $[\tau, T]$. The remaining three bounds specify that the signal on the wire settles within $[h, H]$ after an edge is created by the encoder.[3]

Informally, the safety property we want to check is that the decoder correctly decodes encoded bits from the signal. To formally define the above described informal property, we use the safety property monitor SPM (Figure 3). SPM has a FIFO buffer of size two, representing encoded bits in the signal. When the encoder performs Edge0 or Edge1S, SPM stores a corresponding bit (0 or 1) to its buffer. When the decoder decodes the signal, SPM removes the first bit in its buffer, and compares this bit with the decoded bit by the decoder. SPM sets its decoding_error flag to be true when the buffered bit (which might be a special empty symbol when the buffer is empty) and the decoded bit do not match. SPM sets its buffer_overflow flag to true when the buffer overflow occurs, and sets its buffer_underflow flag to true when the buffer underflow occurs. We say that the protocol is correct under a certain event-order constraint if any of the above described three flags are not raised in all execution of the protocol under the event-order constraint.

4.2 Parameter Constraint Synthesis of BMP Using EOA

In this section, we present parameter constraint synthesis of BMP using EOA.

---

[2] We are treating Decode(0) and Decode(1) (representing decoding a '0' and a '1', respectively) as the same symbol.

[3] The authors of [3] uses different parameters from ours. Namely, they explicitly model clocks in the system (for example, m_1 becomes $mark \cdot min$, where min is the lower bound for the clock cycle, and δ becomes $2min$.) We can retrieve the same clock-cycle information by using equalities between parameters. Therefore, we call our model fully parameterized.

```
EncoderState = enumeration of
                  sending0, sending1S, sending1T
Automaton Encoder(m_1, M_1, c, C: Real) where
   0 ≤ m_1 ≤ M_1 ∧ 0 ≤ c ≤ C

   signature
      output Edge1S, Edge1T, Edge0S
   states
      senderPC: EncoderState :=
                  choose(sending0, sending1S);
   transitions
      output Edge1S
         pre senderPC = sending1S;
         eff senderPC := sending1T;
      output Edge1T
         pre senderPC = sending1T;
         eff senderPC := choose(sending0, sending1S);
      output Edge0
         pre senderPC = sending0;
         eff senderPC := choose(sending0, sending1S);
   bounds:
      b(Edge1S, {Edge1T}) = [m_1, M_1];
      b(Edge1S, {Edge0, Edge1S}) = [c, C];
      b(Edge0S, {Edge0, Edge1S}) = [c, C];
```

Fig. 2. Encoder automaton of the biphase mark protocol

```
BitWithBottom = enumeration of 1, 2 ⊥
Automaton SPM
   signature
      input   Edge1S, Edge0S, Decode(decodedBit:Bit)
   states
      buffer1: BitWithBottom := ⊥
      buffer2: BitWithBottom := ⊥
      decoding_error: Boolean := false
      buffer_overflow: Boolean := false
      buffer_underflow: Boolean := false
   transitions
      input Edge0
         if buffer2 ≠ ⊥ then buffer_overflow := true endif;
         if buffer1 = ⊥ then buffer1 := 0
         else buffer2 := 0 endif;
      input Edge1S
         if buffer2 ≠ ⊥ then buffer_overflow := true endif;
         if buffer1 = ⊥ then buffer1 := 1
         else buffer2 := 1 endif;
      input Decode(decodedBit)
         if buffer1=⊥ then buffer_underflow := true endif;
         if buffer2 ≠ ⊥ and buffer2 ≠ decodedBit ∨
            buffer2 = ⊥ and buffer1 ≠ decodedBit
            then decoding_error := true endif;
         if buffer2 = ⊥ then buffer1 = ⊥ endif;
         buffer2 := ⊥;
```

Fig. 3. Safety property monitor for the biphase mark protocol

```
Bit = enumeration of 0, 1;
SignalVoltage = enumeration of HIGH, LOW;
DecoderState = enumeration of detecting, decoding;
flip(v:SignalVoltage): SignalVoltage = if v = HIGH then
LOW else HIGH endif;

Automaton Decoder(δ, Δ, τ, T, h, H: Real) where
   0 ≤ δ ≤ Δ ∧ 0 ≤ τ ≤ T ∧ 0 ≤ h ≤ H

   signature
      input   Edge1S, Edge1T, Edge0S
      output  Settle
      output  Detect(succeed: Boolean)
      output  Decode(decodedBit: Bit)
   states
      decoderState: DecoderState := detecting;
      signalVoltage: SignalVoltage := LOW;
      signalSettled: Boolean := true;
      oldVoltage: SignalVoltage := LOW;
   transitions
      input Edge1S
         eff signalVoltage := flip(signalVoltage);
            signalSettled := false;
      input Edge1T
         eff signalVoltage := flip(signalVoltage);
            signalSettled := false;
      input Edge0
         eff signalVoltage := flip(signalVoltage);
            signalSettled := false;
      output Settle
         pre signalSettled = false
         eff signalSettled := true
      output Detect(succeed)
         pre decoderState = detecting ∧
            let sampledVoltage: SignalVoltage =
               if signalSettled then signalVoltage
                     else choose(HIGH,LOW) endif in
            succeed = (oldVoltage = sampledVoltage);
         eff oldVoltage = if succeed then flip(oldVoltage)
                     else oldVoltage endif;
            decoderState = if succeed then decoding
                  else detecting endif;
      output Decode(decodedBit)
         pre decoderState = decoding ∧
            let sampledVoltage: SignalVoltage =
               if signalSettled then signalVoltage
                     else choose(HIGH,LOW) endif in
            decodedBit =
               if oldVoltage = sampledVoltage then 0
                  else 1 endif;
            decoderState = detecting;
   bounds:
      b(⊥, {Detect(true), Detect(false)}) = [δ, Δ];
      b(Detect(false),{Detect(true),Detect(false)})=[δ, Δ];
      b(Decode, {Detect(true), Detect(false)}) = [δ, Δ];
      b(Detect(true), {Decode}) = [τ, T];
      b(Edge1S, {Settle}) = [h, H];
      b(Edge1T, {Settle}) = [h, H];
      b(Edge0, {Settle}) = [h, H];
```

Fig. 4. Decoder automaton of the biphase mark protocol

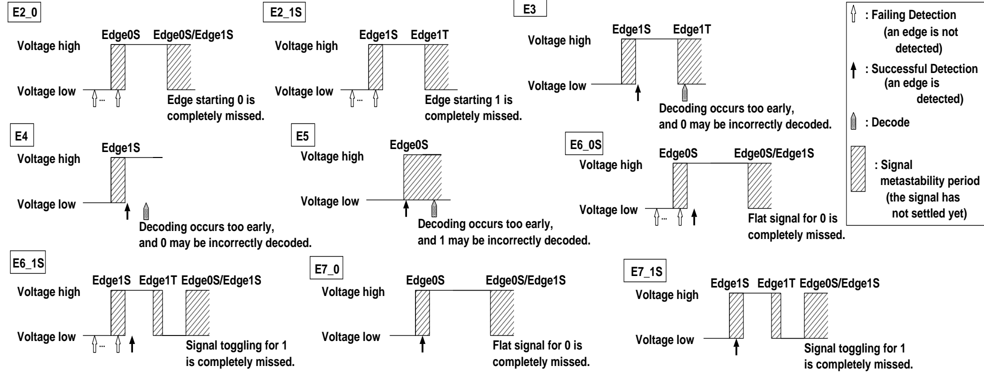

Fig. 5. Bad scenarios of a biphase mark protocol

Step 1. Identifying bad event orders: We first need to identify bad event orders in the protocol execution that would lead to safety property violation. Here we illustrate how the user can identify bad event orders by model-checking the untimed model of BMP. If the user does not know a clue what kind of event orders exist, then he/she can start model-checking the untimed model without any event-order assumption. In this case, he/she obtains the following counterexample: Edge1S - Detect(true) - Decode(0). This execution is bad since decoding is performed too fast and therefore a zero is decoded even though the encoder is sending a one. By excluding only this exact sequence cannot prevent similar scenarios from occurring. This is because an arbitrary number of Detect(false) can appear before Detect(true), and a Settle action can appear any time after Edge1S. Therefore, the user must specify Edge1S - Detect(true) - Decode(0): insert {Detect(false)} in [1,2]; {Settle} in[1,3], in order to exclude all similar executions. (The above event order with an IES is actually E_4 that we describe later in this section.) With an event-order constraint for the above event order, the user obtains the following counterexample: Edge0 - Edge0 - Edge 1S. This execution is bad since the SPM buffer overflows. We can actually observe that this execution is bad also in terms of decoding: the decoder completely misses the first Edge0. Thus the prefix Edge0 - Edge 0S of the counterexample is a bad event order. There are actually two more similar scenarios: Edge0 - Edge 1S (Edge0 is missed by the decoder) and Edge1S - Edge 1T (Edge1S is missed). Therefore, the user wants to specify all of the above three event orders as bad orders. The user can continue identifying bad scenarios until he/she succeeds in model-checking.

We repeated the process of bad event order identification described above, and found nine bad scenarios. Now we list all bad scenarios and explain why they are bad.

The first scenario is the same as the second bad scenario described above.

$E1_1$: Edge0-Edge0 $E1_2$: Edge0-Edge1S $E1_3$: Edge1S-Edge1T

The rest of the scenarios are depicted in Figure 5.

$E2_0$ and $E2_1S$ describe similar scenarios: in these scenarios, the edge starting 0 or 1, respectively, is completely missed because the next edge occurs before the decoder observes the edge.

$E2_0$: Decode-Edge0-Settle-Edge0: insert {Detect(false)} in [1,3]

$E2_1S$: Decode-Edge1S-Settle-Edge1T: insert {Detect(false)} in [1,3]

Detect(false) is inserted in the above event orders since it can possibly occur unbounded number of times before an edge settles.

We also have slightly different versions of the above two event orders. These different versions basically describes the same situation as the originally described two. For example, $E2_0$ can end with Edge1S instead of Edge0, as described in Figure 5. The first Decode in $E2_0$ and $E2_1S$ can be replaced by $\perp$ symbol, representing the same situation as the original one just after the system execution starts (not after at least one Decode have already been performed). Therefore, we had four versions (including the one represented above) for $E2_0$ and two versions for $E2_1S$.

$E3$ describes the case that the sampling distance of the decoder is so short that the decoder decodes the signal before the signal settles after toggling for a '1'.

$E3$: Edge1S-Detect(true)-Edge1T-Decode:
insert {Detect(false)} to [1,2]; {Settle} to [1,3]

$E4$ describes the case that a signal toggling for a '1' has not been done yet when the sampling distance for the decoder elapses, and thus a '0' is incorrectly decoded.

$E4$: Edge1S-Detect(true)-Decode: insert {Detect(false)} in [1,2]; {Settle} in [1,3]

$E5$ describes the case that the sampling distance is so short that the decoding for a '0' is done before the signal settles, and thus a '1' can be incorrectly decoded.

$E5$: Edge0-Detect(true)-Decode: insert {Detect(false)} in [1,2]

$E6_0$, $E6_1S$, $E7_0$ and $E7_1S$ describe similar situations: the sampling distance is long relative to the cell size, and thus decoding for the current has not been done before the next edge occurs.

$E6_0$: Decode-Edge0-Settle-Detect(true)-Edge0: insert {Detect(false)} in [1,3]
$E6_1S$: Decode-Edge1S-Settle-Detect(true)-Edge1T-Edge0:
insert {Detect(false)} to [1,3]; {Settle} in [5,6]
$E7_0$: Edge0-Detect(true)-Settle-Edge0: insert {Detect(false)} in [1,2]
$E7_1S$: Edge1S-Detect(true)-Settle-Edge1T-Edge0:
insert {Detect(false)} to [1,2]; {Settle} in [4,5]

As in the case of $E2$ event orders, $E6_0$, $E6_1S$, $E7_0$, and $E7_1S$ also have slightly different versions that describes the same situation – Edge1S instead of Edge0 at the end of $E6$'s and $E7$'s, $\perp$ instead of Decode in $E6_0$ and $E6_1S$. Therefore, we had four event orders for $E6_0$ and $E6_1S$, respectively, and two for $E7_0$ and $E7_1S$, respectively.

In total, we had 24 event orders (3 $E1$, 6 $E2$, 1 $E3$, 1 $E4$, 1 $E5$, 8 $E6$, and 4 $E7$).

We were able to combine event order monitors for multiple event orders into one monitor (for example, we can easily construct a monitor that monitors four event orders of the type of $E6_0$ at the same time). We had 11 monitors in total for 24 event orders.

We confirmed by untimed model-checking that the safety property described earlier in this subsection holds at any reachable state of our model under the assumption that the above specified event orders do not occur in system executions. We also examined how many encoded bits can exist on the wire as the same time (by counting the number of bits in the buffer of SPM), and confirmed that under the same ordering assumption, only one bit is encoded on the wire at the same time.

Step 2. Deriving timing parameter constraints using METEORS: We specified in our tool METEORS all of Detect(false) events in IES's of the specified event orders as those that must be decomposed. METEORS derived (after an automatic linear arithmetic simplification) a constraint, which is a conjunction of the following three inequalities.

(1). $m_1 > H + \Delta$; (2). $\tau > M_1 + H$; and (3). $c > H + \Delta + T$.

These three inequalities are equivalent to the three constraints manually derived in [3]. Therefore, though we needed to identify all bad scenarios (in terms of event orders), we obtained a result equivalent to [3], without manually deriving parameter constraints from bad scenarios, and safety property verification under the event order assumption was conducted automatically using a conventional untimed model-checker built in SAL.

METEORS also produced the list of constraints that are sufficient to exclude each of the specified bad event orders. Table 1 shows the result.

$E1_1$, $E1_2$	$c > H$	(subsumed by $c > H + \Delta + T$)
$E1_3$	$m_1 > H$	(subsumed by $m_1 > H + \Delta$)
$E2_0$ (all four versions)	$c > H + \Delta$	(subsumed by $c > H + \Delta + T$)
$E2_1S$ (both versions)	$m_1 > H + \Delta$	
$E3$	$\tau > M_1 + H$	
$E4$	$\tau > M_1$	(subsumed by $\tau > M_1 + H$)
$E5$	$\tau > H$	(subsumed by $\tau > M_1 + H$)
$E6_0$ (all four versions) $E6_1S$ (all four versions)	$c > H + \Delta + T$	
$E7_0$ (both versions) $E7_1S$ (both versions)	$c > H + T$	(subsumed by $c > H + \Delta + T$)

Table 1. Sufficient constraints to exclude each bad event order

The whole synthesis process took us about 12 hours: 3-4 hours to model BMP in SAL and find modeling (not design) bug in the model; 5-6 hours to identify the bad event orders and writing monitors for them; and 2-3 hours to code up the event orders as Python objects to be used by METEORS and decompose event orders to obtain reasonable constraints (this time does not include the time we took to learn the basic knowledge of how BMP operates). All computer experiments are conducted on a machine with Intel Core$^{\mathrm{TM}}$2 Quad at 2.66 GHz and 2 GB memory. Each untimed model-checking for bad-event-order identification using SAL (as well as the successful model-checking) took less than one second.[4] The computation time for timing synthesis using METEORS took less than one second, including simplification.

4.3 Related Work and Comparison of Approaches

In this subsection, we compare case studies of BMP using other approaches with ours.

Fully Automated Timing-Parameter Constraint Synthesis Using Model-Checker: A first attempt of parametrically model-checking BMP using HYTECH was reported by Ivanov and Griffioen [8]. They succeeded to verify a model of BMP, but the model was restrictive compared to our model. (for example, sampling of the signal by the decoder was only allowed at particular time points). In [7] by Henzinger et al., the application of HYTECH to BMP is briefly reported. They succeeded to conduct partial timing synthesis (they had to fix a subset of parameters). To do so, they had to modify a model of [3] so that only successful detections of the edge appear in the execution. This modification prevented the model from non-terminating fix-point calculation due to unbounded number of failing detections.

Parameterized Verification Using Inductive Reasoning: Parameterized verification is a different problem from parameter-constraint synthesis, since the user has to provide parameter constraints first. However, since our approach also needs human interaction in order to specify bad event orders, it is interesting to compare the results using the inductive-reasoning approach. In [3], the authors presented parametric verification of a

[4] We are stating the computation time not to compare it with a fully automatic approach (since our approach needs bad event order identification by the user), but to show that adding an event-order monitor does not significantly degrade the speed of model-checking.

biphase mark protocol using the UPPAAL model-checker and the PVS theorem prover. They first used UPPAAL to identify bad scenarios by using several fixed combinations of parameters. They then identified three bad scenarios, depicted a diagram of them, and manually derived linear inequality constraints from the diagram. Then, to verify the correctness of the system under the derived constraints, they conducted a mechanical theorem-proving using PVS [16], by translating the UPPAAL model into PVS code. Though they succeeded in proving the correctness of the protocol under the derived constraints, they required 37 inductive invariants, consisting of several group of invariants that they needed to prove together. Their first verification attempt needed more than 4000 steps of human interactions with PVS. From our experience of a similar type of problem – inductive proof of safety properties for an I/O automaton (thought not timed) using PVS [17], we estimate that the total verification process took them considerably long time, probably in the order of one month.

In [6], Brown and Pike presented semi-automated parametric verification of a biphase mark protocol using the SAL model-checker [15] and the Calendar Automata approach ([18, 19]). By using the Calendar Automata approach, the user can embed the timing constraints of the system behavior in the real-time system using what the authors of [18] call the "calendar" which specifies the time of occurrence of events in the future. Since the model contains uninterpreted constants (timing parameters), the user cannot directly model-check this model, but the verification has to be done by finding inductive properties (though the proving process is completely automated). Brown and Pike [6] manually derived timing parameter constraints in a way similar to [3]. They then had to come up with five supporting inductive lemmas in order to prove the correctness of the protocol under the derived constraints.

5 Conclusion

In this paper, we reported timing-parameter-constraint synthesis of the biphase mark protocol (BMP) using event order abstraction (EOA). BMP has repetitions of events with timing constraints (repeated detections of an edge) and 10 parameters, and these aspects of BMP disable a direct application of existing timed/hybrid model-checkers. By using EOA, we successfully synthesized the parameter constraints equivalent to those manually derived in [3]. The process was machine assisted in two ways: 1. Safety property verification under identified event order assumption was automatically conducted by the untimed model-checker built in SAL; and 2. Parameter constraint was automatically derived by out tool METEORS using event order information from the first step.

We compared the EOA approach with the inductive-reasoning approach, which is the only other approach that is successfully applied to BMP. For the inductive-reasoning approach, the user first needs to identify bad event orders (which are not necessarily all bad orders), and then manually derive sufficient timing-parameter constraints to exclude all bad scenarios. In addition, the properties to be proved or checked must be strengthened or the user needs auxiliary properties to be proved or checked, so that the reasoning becomes inductive. In general, constructing inductive properties requires human insights and some prior training on the inductive-proof methodology. Indeed, the lemmas presented in [6] include non-trivial inequalities over variables of the calendar. Identifying bad event orders is, in contrast, closer to constructing informal operational arguments, and needs less training than constructing inductive properties.

We consider that identifying bad event orders is useful not only for the verification/synthesis process of EOA, but also for implementation engineers to understand

what kind of undesirable scenarios can occur in the underlying system/protocol when parameters are badly tuned. With our tool METEORS, the user can also know what constraint is sufficient to exclude each of the bad scenarios.

For future work, we want to seek a way of automating identification of bad event orders. This automation requires the tool to find subsequences of counterexamples that are sufficient to exclude all bad executions, yet have enough information to derive reasonable constraints.

Acknowledgment: First of all, I thank my supervisor, Prof. Nancy Lynch, for her patient guidance and fruitful advice on this research work. I also thank anonymous reviews for their helpful comments.

References

1. Moore, J.S.: A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. Formal Aspects of Computing **6**(1) (1994) 60–91
2. Umeno, S.: Event order abstraction for parametric real-time system verification. In: EMSOFT 2008: The 8th ACM & IEEE International Conference on Embedded Software. (2008) 1–10 A technical report version appears as MIT-CSAIL-TR-2008-048, Massachusetts Institute of Technology, July, 2008.
3. Vaandrager, F.W., de Groot, A.: Analysis of a biphase mark protocol with UPPAAL and PVS. Formal Asp. Comput. **18**(4) (2006) 433–458
4. Zhang, D., Cleaveland, R.: Fast on-the-fly parametric real-time model checking. In: Proceedings of the 26th IEEE Real-Time Systems Symposium. (2005) 157–166
5. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc. (1996)
6. Brown, G.M., Pike, L.: Easy parameterized verification of biphase mark and 8N1 protocols. In: Proc. of TACAS 2006. Volume 3920 of Lecture Notes in Computer Science., Springer (2006) 58–72
7. Henzinger, T., Preussig, J., Wong-Toi, H.: Some lessons from the HYTECH experience. In: Proc. of the 40th Annual Conference on Decision and Control, IEEE Computer Society Press (2001) 2887–2892
8. Ivanov, S., Griffioen, W.: Verification of a biphase mark protocol. Technical report (1999)
9. Hune, T., Romijn, J., Stoelinga, M., Vaandrager, F.W.: Linear parametric model checking of timed automata. In: Tools and Algorithms for Construction and Analysis of Systems. (2001) 189–203
10. Wang, F.: Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures. Transactions on Software Engineering **31** (2005) 38–51
11. Annichini, A., Bouajjani, A., Sighireanu, M.: TReX: A tool for reachability analysis of complex systems. In: Computer Aided Verification. (2001) 368–372
12. Spelberg, R., Toetenel, W.: Parametric real-time model checking using splitting trees. Nordic Journal of Computing **8** (2001) 88–120
13. Collomb-Annichini, A., Sighireanu, M.: Parameterized reachability analysis of the ieee 1394 root contention protocol using trex. In: RT-TOOL 2001. (2001)
14. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurent Systems: Specification. Springer-Verlag (1993)
15. de Moura, L.M., Owre, S., Rueß, H., Rushby, J.M., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: Proc. of CAV 2004. Volume 3114 of Lecture Notes in Computer Science., Springer (2004) 496–500
16. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In Kapur, D., ed.: 11th International Conference on Automated Deduction (CADE). Volume 607 of Lecture Notes in Computer Science., Saratoga, NY (1992) 748 – 752
17. Umeno, S., Lynch, N.: Proving safety properties of an aircraft landing protocol using I/O automata and the PVS theorem prover: a case study. In: FM 2006: Formal Methods. Volume 4084 of Lecture Notes in Computer Science., Springer (2006) 64 – 80

18. Dutertre, B., Sorea, M.: Timed systems in SAL. Technical Report SRI-SDL-04-03, SRI International (2004)
19. Dutertre, B., Sorea, M.: Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In: Proc. of FORMATS/FTRTFT 2004. Volume 3253 of Lecture Notes in Computer Science., Springer (2004) 199–214