Event Order Abstraction for Parametric Real-Time System Verification [*][†]

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

Abstract

We present a new abstraction technique, *event order abstraction* (EOA), for parametric safety verification of real-time systems in which "correct orderings of events" needed for system correctness are maintained by timing constraints on the systems' behavior. By using EOA, one can separate the task of verifying a real-time system into two parts: 1. Safety property verification of the system given that only correct event orderings occur; and 2. Derivation of timing parameter constraints for correct orderings of events in the system.

The user first identifies a candidate set of bad event orders. Then, by using ordinary untimed model-checking, the user examines whether a discretized system model in which all timing constraints are abstracted away satisfies a desirable safety property under the assumption that the identified bad event orders occur in no system execution. The user uses counterexamples obtained from the model-checker to identify additional bad event orders, and repeats the process until the model-checking succeeds. In this step, the user obtains a sufficient set of bad event orders that must be excluded by timing synthesis for system correctness.

Next, the algorithm presented in the paper automatically derives a set of timing parameter constraints under which the system does not exhibit the identified bad event orderings. From this step combined with the untimed model-checking step, the user obtains a sufficient set of timing parameter constraints under which the system executes correctly with respect to a given safety property.

We illustrate the use of EOA with a train-gate example inspired by the general railroad crossing problem [13]. We also summarize three other case studies, a biphase mark protocol, the IEEE 1394 root contention protocol, and the Fischer mutual exclusion algorithm.

Keywords: Parametric verification, event-based approach, automatic timing synthesis, counter-example guided abstraction refinement (CEGAR)

[*]This work is supported by NSF Award 0702670.
[†]This paper is a refined version of the paper presented in EMSOFT 2008 [30] with a slight typo-fixing in Figure 2 and a revision in Section 7.2.

1 Introduction

In a typical real-time system, timing constraints on the system's behavior are used to ensure its correctness. Such a system is often modeled by using a set of *timing parameters*, rather than using concrete timing constants (for example, [25, 27, 13]). These parameters specify, for instance, bounds on the duration between two specific events in a system execution or certain delays, such as message delivery times.

Typically, only a subset of possible parameter combinations in the entire parameter space satisfies correctness of such a system. A verification engineer or researcher typically follows one of the following two approaches to formally verify such a system: 1. (*Fixed-parameter verification*) By fixing all timing parameters in the system, he/she reduces the system model to a more tractable one such as an Alur-Dill timed automaton [1] and model-checks the reduced system (using UPPAAL [20] or KRONOS [34], for instance [12, 6, 22]); or 2. (*Parametric verification*) he/she treats the timing parameters as uninterpreted constants, finds an appropriate set of constraints for the parameters, and manually proves or mechanically checks correctness under the constraints [25, 32, 35].

The second approach is attractive in the sense that if we can obtain a positive verification result by this approach, then we have a concrete set of constraints on the timing parameters for the system to be correct, and may give an implementation engineer more freedom of choice, than fixed-parameter verification.

The user can experiment with several instances of the first verification approach using multiple parameter combinations, and then can try to figure out possible correlations between parameters in order for the system to be correct (for example, [27] uses this approach). However, these experiments by themselves never become exhaustive if the number of possible parameter combinations is infinite (for example, a parameter can be real-valued, or an integer but unbounded). Thus we need a more intelligent approach for completely parametric verification.

Another important challenge, in addition to time-parametric verification, is *timing synthesis* of a time-parametric model. For timing synthesis, one tries to derive, in a systematic way, a sufficient set of timing parameter constraints under which the system executes correctly. Automatic timing synthesis is considered to be an even harder problem than automatic time-parametric verification since an algorithm or a tool is not a priori given a set of timing constraints by the user, but has to derive constraints by itself. A classical undecidability result about parametric timed automata by Alur et al. [2] implies that a completely automatic timing synthesis does not terminate in general.

In this paper, we present a new abstraction technique, *event order abstraction* (EOA), for parametric safety verification of the subclass of real-time systems in which *correct orderings of events* maintained by timing constraints on the systems' behavior are critical for correctness (for example, a biphase mark protocol [25], the Fischer mutual exclusion algorithm ([21], Section 24.2), and the IEEE 1394 root contention protocol [27]). By using EOA, one can separate the task of verifying a safety property of a system into two parts: 1. Safety property verification of the system given that only correct event orderings occur; and 2. Derivation of timing parameter constraints for correct orderings of events in the system.

To use EOA, the user models a real-time system by using the *time-interval automata* (TIA) framework, which is an extension of the I/O automata framework [21], and can express certain restricted class of timed I/O automata [19]. By using the TIA framework, the user can specify lower and upper bounds on the time interval between a specific event and a set of possible events that follow. The framework has a certain structure that is suitable for a mechanical timing constraint derivation scheme presented in this paper.

A parametric verification of a real-time system using EOA is conducted in the following steps. First step is identification of "bad" event orders. The user proposes a candidate set of bad event orders that he/she wants to exclude from the system executions by timing synthesis. The user then model-checks a safety property of interest on a *discretized model* of the underlying TIA, under the assumption that the model does not exhibit the proposed bad event orders A discretized model of a TIA is simply an ordinary untimed I/O automaton that does not have any timing constraints as the original TIA does. If the model-checking is completed with a positive answer, the user has obtained a set of bad event orders that he/she needs to exclude. Otherwise, the user uses counterexamples obtained from the model-checking to extract additional bad event order, and repeats the same process until he/she successfully model-checks the discretized model.

The user expresses bad event orders by a simple language that can express a sequential order of events and some types of repetition of events. He/she typically needs to apply human insight to extract from the counterexample a bad event order expressed in a concise way, and this is why we have manually identified bad event orders for the case studies presented in the paper.

Model-checking under a specific event order assumption can be carried out in the following two steps. The user first constructs a monitor that raises a flag when one of the identified bad event orders is exhibited. Then he/she model-checks the discretized model with this monitor under the assumption that the monitor does not raises a flag (in Linear Temporal Logic (LTL) [23], this condition can be represented by: $\Box(\neg \mathsf{Monitor.flag}) \Rightarrow \Box(\neg \mathsf{DiscretizedModel.propertyViolated})$). We used the SAL model-cheker [9] in this paper. We manually constructed monitors since the construction was straightforward for the presented case studies (we are planning to develop an automatic monitor construction tool). Since we successively refine the underlying discretized model (by refining the bad order assumption) from a counterexample obtained from model-checking, EOA can be regarded as a *counterexample guided abstraction refinement* (CEGAR) technique [7].

Next, by an algorithm that we present in the paper, the user automatically derives timing parameter constraints under which the system exhibits none of the identified bad event orders. From this step, the user obtains sufficient timing constraints under which the system executes correctly with respect to a given safety property.

Related work: Some of the existing timed model-checkers (HyTech [14], RED [33], TReX [3], LPMC [28], and an extension of UPPAAL [18]) allow *automatic synthesis* of timing parameters for a specified desirable property of a given system: these tools automatically derive a set of constraints on timing parameters for the system to satisfy a given property. However, termination is in general not guaranteed for these model-checkers.

The main differences of EOA from the existing automatic timed model-checkers listed above are the following four.

First, to use EOA, the user has to provide a set of bad event orders to be excluded in the system by timing synthesis. Timed model-checkers mentioned above does not need such inputs.

Second, EOA can treat a class of systems that may exhibit an unbounded number of repetitions of events. The existing parametric model-checkers listed above use symbolic reachability analysis of states symbolically represented by linear logic expressions. Thus, if an underlying parametric model has an unbounded loop that involves evolution of continuous variables, then this reachability analysis does not terminate, and therefore the verification attempt fails (for example, in [14], Section 4.2, the authors stated that they had to modify a model of a biphase mark protocol so that it exhibits no unbounded loop). In EOA, by using a language construct that represents an unbounded number of repetitions of events, the user can handle this kind of system.

Third, when doing successive refinements by using EOA, each abstraction in a refinement step is a completely untimed transition system (an ordinary I/O automaton with ordering constraints). Thus the user can directly employ existing verification techniques for untimed transition systems.

Fourth, EOA does not suffer from the "dimensionality problem" as much as the timed model-checkers listed above do. Automatic timing synthesis using the above listed model-checkers rapidly becomes intractable as the number of parameters grows ([14], Section 5. Lessons learned). This problem is called the "dimensionality problem", and is regarded as one of the main bottle necks of the time-parametric model-checkers. With EOA, timing synthesis is handled separately from model-checking – the tool derives timing parameter constraints from identified event orders just with information about time bounds between events, and does not use any information about the state transition structure of the system. This synthesis process does not use a fixed-point computation as timed model-checkers do, and thus does not need linear logic simplification for termination[1]. Instead, as we present in Section 6, timing synthesis is done by a straightforward search within a certain space inferred by specified event orders. In all case studies summarized in this paper, the search spaces were small. Indeed, the train-gate example that we use to illustrate EOA throughout the paper has ten parameters, and the timing synthesis for it from specified event orders took less than one second.

Frehse, Jha, and Krogh [11] presented a CEGAR-based approach for automatically synthesizing parameter constraints of linear hybrid automata (LHA) [15]. Though this work is independently done from our work, the approach is similar to ours in that it uses discrete abstraction of the underlying system to obtain counterexamples, and then synthesize the timing (continuous) parameter constraints to exclude the obtained counterexamples. The main differences between their approach and our approach are the following three: 1. Their approach automatically identifies bad event sequences; 2: Their approach does not treat a repetition of events as our approach does (Treating repetitions is crucial to verify certain examples such as the train-gate example in this paper and a biphase mark protocol, for which meaningful parameter constraints can be obtained only by treating repetitive events); 3: Their approach treats LHA, which is more general than TIA. They experimented their approach by a simple car-conflict prevention example, which has only two parameters. The applicability of their approach to a system with a large number of parameters such as the ones in Section 7 is not known.

Several researchers considered digitization of timed transition systems [17, 5, 4, 26]. These techniques could possibly be used to obtain a discrete version of real-time systems for fixed parameters, but as far as we know, an application of the technique to parametric verification has not been studied.

We have developed EOA to fill in the gap between the inductive proof approach and automatic time-parametric model-checking. The inductive proof approach needs human insights into an underlying system to come up with an inductive property, and we believe that identifying bad event orders is more amenable process and requires less training than coming up with inductive properties. On the other hand, automatic time-parametric model-checking may not always scale to a system with a considerable number of timing variables and parameters, as we described earlier.

When automatic time-parametric model-checker does not scale, one can try using inductive invariant reasoning or by model-checking using parameter constraints as inputs – these are typically more scalable compared to automatic parameter synthesis tools. To do so, he/she first needs to derive a set of timing parameter constraints under which (he/she believes) the system works correctly. Typically the user performs this derivation by first drawing a process communication diagram that depicts a possible bad scenario, and then manually finding out how to constrain timing parameters to exclude the depicted scenario. This approach is used in [32] to verify a biphase mark protocol, and in [27] for the root contention resolving algorithm of the IEEE 1394 protocol. With EOA, the user can directly make use of these human insights into the bad scenarios, and can also automate the process of deriving timing constraints from the bad scenarios.

The rest of the paper is organized as follows. In Section 2, we introduce a new automata framework, *time-interval automata*. We present the train-gate example, which is inspired by a railroad crossing problem [13], in the TIA setting. We use this example to illustrate the use of EOA throughout the paper. The example is simple compared to an industrial protocol, for example, a biphase mark protocol that we study in Section 7, yet has ten parameters and exhibits an unbounded repetition of events. In Section 3, we explain how the user can formally specify event orders. In Section 4, we demonstrate how the user can conduct the bad-event-order identification step. Section 5 is devoted to presenting the basis for automatic timing constraint derivation. In Section 6, we present a prototype implementation that automatically synthesize timing constraint from given event orders. Section 7 presents case studies of time-parametric verification using EOA. We first present verification of the train-gate example. We also summarize three other case studies, a biphase mark protocol that has been studied in several verification papers (for example, [25, 32]), the IEEE 1394 root contention protocol [27], and the Fischer mutual exclusion algorithm ([21], Section24.2). As a conclusion, in Section 8 we discuss a summary of the paper and possible future work.

2 Time-Interval Automata

The *time-interval automata* (TIA) framework is an extension of the I/O automata (IOA) framework [21]. An I/O automaton is a guarded-command style transition system with distinguished input, output, and internal actions. 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 action a and actions in 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 $t_a + u$.

As we explain in the reminder of this section, a TIA has an explicit structure to specify the time bounds for actions, or events. The automatic derivation scheme we present in Section 5 and also the prototype implementation introduced in Section 6 make use of this explicit structure to conduct a timing synthesis. We compare the relation between the TIA framework and other timed automata frameworks in the related work part in this section.

An *interval-bound map* defined in the following Definition 1

[1]Nevertheless, a linear logic simplification for a derived set of constraints is provided by the prototype tool for user's convenience.

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). Due to space limitation, we cannot show these requirements in this paper. These appear in [31]. 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 [31]).

A composition of multiple TIA is defined in a way similar to that of ordinary I/O automata. 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 [31] for the formal 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 $\{A_i\}_{i\in I}$ (which is an ordinary asynchronous composition with synchronization of input and output actions with the same name [21]), 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).

A TIA must satisfy the *feasibility* condition. Namely, every execution of a TIA must be extended to a time-diverging execution (that is, $\sup_{i\geq 0}\{t_i\} = \infty$). The definition appears in [31]. All composed TIA of case studies in the present paper are feasible.

Definition 5. (Discretized TIA) Given a TIA (A, b), the discretized model of (A, b) is simply an underlying ordinary untimed I/O automaton A.

The set of (untimed) executions of a TIA (A, b) (obtained by ignoring time stamps in timed executions) is contained by the set of executions of its discretized model A, since A does not have any timing constraint. Thus, if A satisfies a safety property under a certain event ordering assumption for its executions, then (A, b) also does so under the same ordering assumption.

Related work of the TIA framework: The timed I/O automata (TIOA) framework [19] is a highly expressive framework with which the user can specify continuous evolution of *analog variables* by using differential equations and inequalities, as well as specifying discrete state transitions as in an ordinary I/O automaton. Indeed, any TIA can be expressed as a TIOA as well. However, a TIOA does not have an explicit time bound structure like a time-interval bound map of a TIA, and thus information about time bound cannot be easily handled by the scheme or the tool presented in the paper (a time lower bound needs to be embedded in the precondition of an action, and an upper bound needs to be expressed by another construct, the *stop-when* statement).

The MMT (time-constrained) automata framework [24] is closely related to the TIA framework. While a TIA specifies time upper and lower bounds on the interval between an event and a set of events that follow, an MMT automaton specifies time upper and lower bounds on the duration that an action in a specific set of actions called a *task* stays enabled. When we define a TIA, for a pair (π, Π) of an action and an action set with a bound defined, we impose constraints on the TIA so that at least one action in Π must be enabled after π and before an action in Π is performed. If we impose the same constraint on an MMT automaton, we have a framework similar to TIA. The timed transition system framework [16] is close to the MMT automata framework, in that the lower and upper time bound on the duration that one transition is enabled can be specified. One main difference between TIA and these two frameworks is that in TIA, the user can use different bounds for the same set of actions depending on which action precedes it. We need this feature to model a biphase mark protocol.

The Alur-Dill timed automata framework [1] is arguably the best known framework to model a real-time system, and is the

```
Automaton Train($r, R, p, P$: Real) where
    $0 \le r \le R \wedge 0 \le p \le P$
    signature
        output Request
        output Pass
    states
        requested: Bool := false;
    transitions
        output Request
            pre ¬requested
            eff requested := true;
        output Pass
            eff requested := false;

    bounds:
        $b(\bot, \{\text{Request}\}) = [r, R]$;
        $b(\text{Pass}, \{\text{Request}\}) = [r, R]$;
        $b(\bot, \{\text{Pass}\}) = [p, P]$;
        $b(\text{Pass}, \{\text{Pass}\}) = [p, P]$;
```

Figure 1: Train automaton

theoretical foundation for timed model-checkers like UPPAAL [20] and KRONOS [34]. This framework can model only a system with fixed timing parameters, but not a time-parametric system.

The parametric timed automata (PTA) framework introduced in [2] is a time-parametric version of the Alur-Dill timed automata framework. In a PTA, the user specifies lower and upper bounds on a time interval in which the automaton stays in a specific *location* (in the Alur-Dill timed automata sense). A TIA can be modeled as a PTA, but time bound for events becomes implicit (unlike the explicit interval-bound map) and thus cannot directly use the automatic timing synthesis scheme presented in the paper.

Example 1. (Time-Interval Automaton). We describe an example of time-interval automata. The example is inspired from railroad crossing problems [13]. The example is constructed from a composition of a train automaton (Figure 1) and a gate automaton (Figure 2). An informal description of the problem we want to solve is the following. A train is about to pass the railroad crossing with a gate. The gate is supposed to be open except for the time when the train passes the crossing, so that cars can cross the railroad. When the train gets close to the crossing, it *requests* that the gate be closed. The gate needs to be closed at the time the train passes the crossing. The railroad actually forms a circle, and thus the train passes the railroad crossing cyclically. After the gate becomes closed, it becomes open after a bounded time interval.[2]

The actions of the Train automaton model actions taken by the train. The Request action represents a close request made by the train to the gate. The Pass action represents an event that the train passes the crossing. The automaton has four bounds

[2]If the reader prefers an example with more digital system flavor than the train-gate example, he/she can regard this example as, for instance, the following single-writer/multi-reader shared variable problem: one writer process (Train) writes to a shared variable (railroad crossing) periodically, and before writing to the variable, it first requests the guardian process (Gate) to lock the variable so that any reader (a car crossing the rail-road) cannot access to the variable while the writer is writing to it.

```
Automaton Gate($\delta, \Delta, \tau, T, c, C$: Real) where
    $0 \le \delta \le \Delta$, $0 \le \tau \le T$, $0 \le c \le C$
    signature
        input Request
        output Close
        output Open
        output Check(result: Bool)
    states
        open: Bool := true;
        train_requested: Bool := false;
        check_succeeded: Bool := false
    transitions
        input Request
            eff train_requested := true;
        output Close
            pre check_succeeded ∧ open
            eff open := false;
        output Open
            pre ¬open
            eff open := true;
                train_requested := false;
                check_succeeded := false;
        output Check(result)
            pre ¬check_succeeded ∧ result = train_requested
            eff check_succeeded := train_requested;

    bounds:
        $b(\bot, \{\text{Check(true)}, \text{Check(false)}\}) = [\delta, \Delta]$;
        $b(\text{Check(false)}, \{\text{Check(true)}, \text{Check(false)}\}) = [\delta, \Delta]$;
        $b(\text{Open}, \{\text{Check(true)}, \text{Check(false)}\} = [\delta, \Delta]$;
        $b(\text{Check(true)}, \{\text{Close}\}) = [\tau, T]$;
        $b(\text{Close}, \{\text{Open}\}) = [c, C]$;
```

Figure 2: Gate automaton

for these two actions. The first one ($b(\bot, \{\text{Request}\}) = [r, R]$) and the second one ($b(\text{Pass}, \{\text{Request}\}) = [r, R]$) say that the Request action will be performed within the time interval $[r, R]$ after the system starts, and every time after the train passes the crossing, respectively. The third bound ($b(\bot, \{\text{Pass}\}) = [p, P]$) and the fourth bound ($b(\text{Pass}, \{\text{Pass}\}) = [p, P]$) say that the Pass action will be performed within the time interval $[p, P]$ after the system starts, and every time after the train passes the crossing, respectively.[3] The gate automaton described in Figure 2 models a gate system that uses a busy-wait loop for checking whether a request has been made. The gate automaton cannot immediately know the arrival of a request. Instead, a request information is stored in a state variable train_requested, and the gate automaton needs to repeatedly check this variable (expressed by a successful check, Check(true), and a failing check, Check(false)). We set the time interval between two repeated checks to be within $[\delta, \Delta]$. Once a check succeeds, the gate automaton stops checking train_requested, but resumes it within $[\delta, \Delta]$ after the gate becomes closed. The gate becomes closed (Close action) within the time interval $[\tau, T]$ after a successful check. The gate becomes open again (Open action) withing the time interval $[c, C]$ after it becomes closed.

The safety property that we want to verify is that the train passes the crossing only when the gate is closed. We use a

[3]We could, for example, think that a train is moving with a bounded velocity within $[v_{min}, v_{max}]$, and the length of the railroad is L. The time bound of $[p, P]$ for the pass event is equivalent to saying that $p = L/v_{max}$ and $P = L/v_{min}$.

monitor automaton SM (safety monitor) that monitors output actions Pass, Open, and Close from Train and Gate, and set its state variable propertyViolated to true if Pass occurs when the gate is open (see [31] for a formal description). The invariant (safety property) we want to check is: for any reachable state of Train||Gate||SM, SM.propertyViolated = false.

3 Specifying Event Orders

In this section, we introduce a formal way of specifying an event order that needs to be excluded for system correctness. We first consider a simple way of specifying an event order, and then extend an event order specification by introducing "don't-care" events. The notion of these "don't-care" events are important in order to treat a repetition of events in a single system (as we will see in the case study for the train-gate example in Section 7.1) and in order to ignore events by a process that is unrelated to a key local behavior in concurrent or distributed systems (as we will see in the case study for the Fischer mutual exclusion in Section 7.4).

An event order (without "don't-care") simply specifies the order of consecutive actions in an execution of a TIA. For example, the event order "Request-Pass" for the automaton (Train||Gate) of Example 1 matches any execution of (Train||Gate) that contains a Request action immediately followed by a Pass action. We give a formal definition of a match between an automaton execution and an event order in Definition 9, after introducing "don't-care" events. An event order may start with a $\perp$ symbol, which specifies that the event order matches a finite *prefix* of an execution of an underlying automaton. In other words, an event order that start with $\perp$ specifies the very first sequence of events that occurs after the automaton starts executing.

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 $\perp$.

Example 2. (Event order). An example of event orders that we want to exclude in Train||Gate||SM of Example 1 is
$\perp$-Check(false)-Request-Check(true)-Pass. In this event order, the gate module first failed to detect a request from the train since a request has not been made yet. After the train makes a request, the gate module succeeds in detecting it. However, the request is detected too late relative to the time for the gate module to close the gate, and consequently the train passes the crossing before the gate becomes closed (that is, before the Close event occurs).

For a system that exhibits an unbounded repetition of events (such as the train-gate example in Example 1 and a biphase mark protocol that we study in Section 7), some event orders to be excluded cannot be represented in a form of a simple event order like the ones we consider above. Consider the event order "$\perp$-Pass" for (Train || Gate). This event order needs to be excluded for an obvious reason: the train passes the crossing even before the train requests that the gate be closed. Considering that the gate is doing a busy-loop checking of a request, this Pass event can possibly be preceded by multiple failing checks (Check(false)). Indeed, since the relation between the frequency of these checks (δ and Δ) and the time when a request is made (r and R) is unknown, the number of possible failing checks that precede the Pass event is unbounded. What we want to do is to *ignore* these failing checks in between $\perp$ and Pass in the event order. By using a regular-expression-like language, this event order can be expressed by "$\perp$-(Check(false))*-Pass", where '$*$' is a symbol of repetition. The following event order using an *ignored event specification* (IES) is more comprehensible when an event is ignored for a specific event-index interval, not just in between two consecutive events: E_2 = "$\perp$-Pass: insert $\{$Check(false)$\}$ to$[0, 1]$". Informally, the ignored event specification (statement after insert)) in the above event order E_2 specifies that when checking a match between an automaton execution and the event order, we ignore in that execution (possibly multiple) occurrences of Check(false) in between the beginning of the execution (e_0) and the first occurrence of Pass (e_1). A formal definition of an IES is as follows.

Definition 7. (Ignored event specification). An ignored event specification (IES) for an event order is in the following form: insert $(Y_m \text{ to } [i_m, j_m])_{m=1}^{r}$, where Y_m represents a set of events that are ignored in the interval between e_{i_m} and e_{j_m}.

To formally define a match between an automaton execution and an event order with an IES, we need an ignored event set I_k^E that represents the set of the ignored events in the interval between the k-th and (k+1)-st events in E ($\perp$ is considered as the zero-th event).

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

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 an 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 *$\alpha$ matches an event order with an IES, $E = e_1 \cdots e_n$:*
insert $(Y_m \text{ to } [i_m, j_m])_{m=1}^{r}$, 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, $1 \le i \le n$, $\pi_{k_i} = e_i$, 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.

We refer to an execution that matches E as an *E-matching execution.*

4 Identifying Bad Event Orders

In this section, we illustrate how the user can extract bad event orders from counterexamples obtained from untimed model-checking of the discretized model.

We use the train-gate example. The safety property we want to check is that the gate is closed whenever the train passes the gate.

We first specified the following set of bad event orders as a candidate[4]:

A_1. $\perp$-Pass : insert $\{\mathsf{Check(false)}\}$ to $[0,1]$
A_2. $\perp$-Request-Pass : insert $\{\mathsf{Check(false)}\}$ to $[0,1]$
A_3. $\perp$-Request-Check(true)-Pass : insert $\{\mathsf{Check(false)}\}$ to $[0,1]$

The above event orders A_1, A_2, and A_3 represent a situation that the train passes the crossing before the gate becomes closed. A_1 specifies a situation that the train passes the gate even before it requests the gate be closed. A_2 specifies a situation that the train has requested the gate be closed, but the gate automaton does not detect a request before the train passes the crossing. A_3 specifies the situation that the gate automaton successfully detects a close request, but the gate does not become closed before the train passes the crossing. Here we used our human insight into the underlying system that an unbounded number of Check(false) events can appear before the Request event.

We manually constructed event order monitors, $\{\mathsf{EOM_i}\}_{i=1}^{3}$, for these event orders, and then model-checked the untimed model under the assumption that the above orders do not appear in system executions. In Linear Temporal Logic (LTL) [23], this condition can be expressed by: $\mathsf{UntimedTrain}||\mathsf{UntimedGate}||\mathsf{SM} \models$
$\Box(\neg\bigvee_{\mathsf{i=1}}^{3} \mathsf{EOM_i.flag}) \Rightarrow \Box(\neg\mathsf{SM.propertyViolated})$. A counterexample that can be obtained from a LTL expression in this form starts with a system execution that leads to a bad state, followed by a *cycle* in which the flags of all monitors never become true. This is because we use the "always" $\Box$ operator for the ordering assumption. The user can basically ignore the cycle part and can just focus on the first part of the counterexample that contains information about a bad event order.

When we model-checked the safety property with the ordering assumption that A_1, A_2, and A_3 do not occur, we obtained the following counterexample execution: Request - Check(true) - Close - Open - Pass, followed by a cycle in which $\bigvee_{\mathsf{i=1}}^{3} \mathsf{EOM_i.flag}$ never becomes true. This execution represents a situation that the gate successfully becomes closed before the train passes the crossing, but becomes open again too fast. Since we knew that multiple Check(false) events could have appeared before the Request event and after the Open event in this execution, we identified the following bad event order.

B_1. $\perp$-Request-Check(true)-Close-Open-Pass :
insert $\{\mathsf{Check(false)}\}$ to $[0,1]$, $\{\mathsf{Check(false)}\}$ to $[4,5]$

[4]Of course, the user could instead start by model-checking the untimed model with no ordering constraint, and build up sufficient event orders. Nevertheless, if the user knows partial information about what bad event orders might be, he/she can use human insight to set up a candidate set of bad orders at the beginning, as in the presented case.

In this way, the user can continue identifying bad event orders using both counterexamples from untimed model-checking and human insight. We present the entire set of bad event orders for the train-gate example in Section 7.1.

5 Deriving Timing Constraints

In this section, we present a scheme to derive a timing parameter constraint to exclude an execution that matches a given event order. The scheme just uses the bound map of an underlying TIA, but not the state-transition structure of it.

Derivation of a timing parameter constraint for a given event order is taken in the following three steps:

1. We enumerate bounds on a pair of events in the event order that are immediately derivable from the bound map b of an underlying TIA and the bound conditions in Definition 3.
2. We combine enumerated individual bounds to form a time bound for larger interval of events in order to derive a meaningful constraint in the next step.
3. We find a matching pair of combined upper bound and lower bound, and then derive a timing constraint.

As we show in Section 6, this scheme forms the basis for the prototype implementation. More specifically, each step of the above described scheme is systematic, and can be easily automated. We present a more detail of each of the steps in the following.

Enumerating bounds: Given an event order E and the bound map b of a TIA, we first enumerate the upper and lower bounds between the time of occurrence of two events in E from the upper and lower bound conditions in Definition 3. The bound sets $U_{i,j}^{E}$ and $L_{i,j}^{E}$ defined in Definitions 10 and 11, respectively, contain upper and lower bounds between the times of occurrence of the actions that match e_i and e_j in E, respectively, that are immediately derivable from the bound map b and the upper and lower band conditions in Definition 3 (the $\perp$ symbol is treated as the zero-th event e_0). The bounds are tagged with the event-index interval for which they are derived. Note that, for a pair (π, Π) of an action and an action set with a bound, an upper bound for an event-index interval $[i,j]$ is found from the fact that the i-th event is π, and an event in Π *does not appear in* $[i,j]$, whereas a lower bound for $[i,j]$ is found from the fact that the i-th event is π, and an event in Π *appears at* the j-th event. This is consistent with the upper and lower bound conditions in Definition 3. For any E-matching execution $\alpha = s_0(\pi_1, t_1)s_1\cdots$, the matched subsequence of actions $\beta = \beta_0\pi_{k_1}\beta_1\cdots\beta_{k_{n-1}}\pi_{k_n}$ (in Definition 9) satisfies $t_{k_j} - t_{k_i} \leq u$ for $(u,[i,j]) \in U_{i,j}^{E}$, and $t_{k_j} - t_{k_i} \geq l$ for $(l,[i,j]) \in L_{i,j}^{E}$ (this fact is proved as a key lemma for Theorem 1 in [31]).

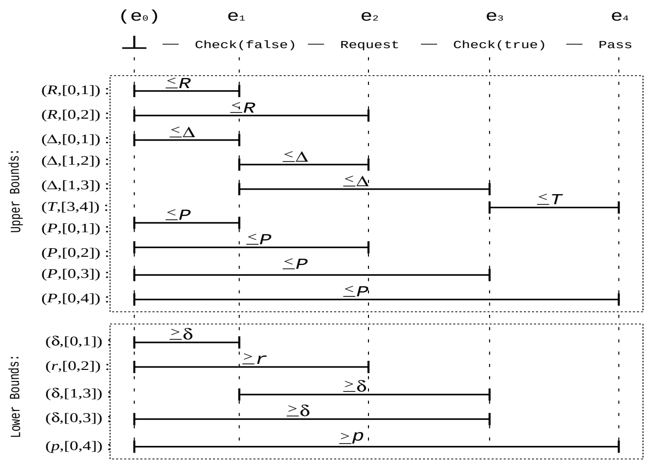

Figure 3: Upper and lower bounds for the event order E_1

Definition 10. (Upper bound set). For i and j, $0 \leq i < j \leq n$,

$$\begin{aligned} U^E_{i,j} = \{(u,[i,j]) \mid u = & upper(e_i,\Pi) \text{ such that} \\ & (e_i,\Pi) \in Domain(upper) \wedge \\ & (j = i+1 \vee \\ & \; e_{i+1}\cdots e_{j-1} \text{ contains no action in } \Pi) \wedge \\ & \cup_{k=i}^{j-1} I^E_k \text{ contains no action in } \Pi.\} \end{aligned}$$

Definition 11. (Lower bound set). For i and j, $0 \leq i < j \leq n$,

$$\begin{aligned} L^E_{i,j} = \{(\ell,[i,j]) \mid \ell = & lower(e_i,\Pi) \text{ such that} \\ & (e_i,\Pi) \in Domain(lower) \wedge \\ & e_j \in \Pi\} \end{aligned}$$

Note that the bound map of an underlying TIA is used only in this first enumeration step.

Example 3. (Upper and lower bound sets). We show an example of $U^E_{i,j}$ and $L^E_{i,j}$. The underlying automaton is Train||Gate||SM discussed in Example 1, the train-gate model with a busy-loop checking. As discussed in Example 2, one of the event order that we want to exclude is $E_1 =$ $\perp$-Check(false)-Request-Check(true)-Pass. Figure 3 depicts the upper bounds in $U^{E_1}_{i,j}$ and lower bounds in $L^{E_1}_{i,j}$.
Upper bound example: We have an upper bound $(R,[0,1])$ for the interval between e_0 ($\perp$) and e_1 (Check(false)) since we have an upper bound $upper(\perp, \{\text{Request}\}) = R$ defined in the bound map, and the event Request is not performed between e_0 and e_1. For a similar reason, we have an upper bound $(R,[0,2])$ between e_0 ($\perp$) and e_2 (Request). The upper bound set $U^{E_1}_{0,1}$ for the interval between e_0 and e_1 is: $\{(R,[0,1]),(P,[0,1]),(\Delta,[0,1])\}$.
Lower bound example: We have a lower bound $(\delta,[1,3])$ for the interval between e_1 (Check(false)) and e_3 (Check(true)) since we have a lower bound
$lower(\text{Check(false)}, \{\text{Check(false)}, \text{Check(true)}\})) = \delta$ defined in the bound map.

Combining bounds: We need a notion of a covering upper bound set and a distributed lower bound set to combine individual bounds in $U_{i,j}$ and $L_{i,j}$, respectively, so that we can synthesize a meaningful timing constraint. Informally, a covering upper bound set U for an event interval Γ is a set of upper bounds such that when we take a union of all intervals that tag upper bounds in U, the union becomes Γ (tagged intervals of upper bounds in U *cover* Γ). A distributed lower bound set L for an event interval Γ is a set of lower bounds such that each interval that tags a lower bound in L is contained in Γ, and all intervals that tag lower bounds in L do not overlap (tagged intervals of lower bounds in L are *distributed* in Γ, without overlapping).

Definition 12. (Covering upper bound set). Consider a set of upper bounds $S = \{(u_k,[i_k,j_k])\}^m_{k=1}$ for a time-interval automaton (A,b) and an event order E (possibly with an IES), where $(u_k,[i_k,j_k]) \in U^E_{i_k,j_k}$ for k, $1 \leq k \leq m$. We say that S *covers the interval between* e_v *and* e_w if for any event pointer p, $v \leq p \leq w-1$, there exists an upper bound $(u_{k_1},[i_{k_1},j_{k_1}]) \in S$ such that $i_{k_1} \leq p$ and $p+1 \leq j_{k_1}$.

Definition 13. (Distributed lower bound set). Consider a set of lower bounds $S = \{(l_k,[i_k,j_k])\}^m_{k=1}$ for a time-interval automaton (A,b) and an event order E (possibly with an IES), where $(l_k,[i_k,j_k]) \in L^E_{i_k,j_k}$ for k, $1 \leq k \leq m$. We say that S *is distributed in the interval between* e_v *and* e_w if the following two conditions hold:

1. For any lower bound $(l_{k_1},[i_{k_1},j_{k_1}]) \in S$, $v \leq i_{k_1}$ and $j_{k_1} \leq w$.
2. For any two lower bounds $(l_{k_1},[i_{k_1},j_{k_1}]),(l_{k_2},[i_{k_2},j_{k_2}]) \in S$, $j_{k_1} \leq i_{k_2}$ or $j_{k_2} \leq i_{k_1}$.

Example 4. (A covering upper bound set and a distributed lower bound set). Let us look at Figure 3 again. The set of upper bounds $\{(R,[0,2]),(\Delta,[1,3]),(T,[3,4])\}$ covers the interval between e_0 and e_4 ($[0,2]\cup[1,3]\cup[3,4] = [0,4]$). Each lower bound by itself constructs a lower bound set that is distributed in the interval between e_0 and e_4, but any set with two or more lower bounds is not distributed in the same interval, since we have some overlap of the intervals for which the lower bounds are defined.

Deriving bounds: The following Theorem 1 implies that if we find a covering upper bound set and a distributed lower bound set for the same interval, then we can obtain the timing constraints by the third condition in the theorem (the sum of the upper bounds is strictly less than the sum of the lower bounds). A formal proof of this theorem appears in [31].

Theorem 1. *Consider an event order E, possibly with an IES. A time-interval automaton (A,b) exhibits no E-matching execution if there exists a set of upper bounds $U = \{(u_m,[i_m,j_m])\}^p_{m=1}$ where $(u_m,[i_m,j_m]) \in U^E_{i_m,j_m}$, a set of lower bounds $L =$ $\{(l_r,[i_r,j_r])\}^q_{r=1}$ where $(l_r,[i_r,j_r]) \in L^E_{i_r,j_r}$, and two events e_v and e_w such that the following three conditions hold:*

1. U covers the interval between e_v and e_w.

2. L is distributed in the interval between e_v and e_w.

3. $\sum_{m=1}^{p} u_m < \sum_{r=1}^{q} l_r$.

Example 5. (Timing constraint derivation for an event order without an IES). Again, consider the event order depicted in Fig. 3. As discussed in Example 4, the upper bound set $\{(R, [0, 2]), (\Delta, [1, 3]), (T, [3, 4])\}$ covers the interval between e_0 and e_4. In addition, the lower bound set $\{(p, [0, 4]\}$ is distributed in the same interval. From Theorem 1, if $p > R+\Delta+T$, then (Train || Gate) exhibits no E_1-matching execution.

Example 6. (Timing constraint derivation for an event order with an IES). Consider the event order E_2 = "$\perp$-Pass: insert Check(false) to$(0, 1)$". We have a lower bound $lower(\perp, \{\mathsf{Pass}\}) = p$, and $\perp$ appears at e_0 and Pass at e_1. Thus we have a lower bound p between e_0 and e_1 (from Definition 11). We have an upper bound $upper(\perp, \{\mathsf{Request}\}) = R$ defined for Train||Gate, and the Request event is not ignored in the interval between e_0 ($\perp$) and e_1 (Pass) – only Check(false) is ignored. Thus we have a valid upper bound R between e_0 ($\perp$) and e_1 (Pass). Therefore, we can derive a constraint $p > R$, which imposes an order constraint that a Request event must occur before a Pass event. On the other hand, though we have an upper bound
$upper(\perp, \{\mathsf{Check(true)}, \mathsf{Check(false)}\}) = \Delta$, we cannot derive an upper bound Δ between e_0 and e_1, since Check(false) is ignored in that interval. Therefore, we cannot derive a constraint $p > \Delta$. Indeed, the above constraint does not exclude E_2, since the constraint just imposes that the very first Check event after each Open event must occur before Pass of that round.

6 Implementation

We have implemented in Python a prototype of a timing constraint derivation tool (METEORS: MEchanical Timing / Event-ORder Synthesizer, version 0.1), based on the scheme described in Section 5. The problem that the implemented prototype tool solves is as follows. The user gives the tool the set of time bounds defined in an underlying TIA for which he/she wants to derive a timing parameter constraint. Then the user gives the tool (typically multiple) bad event orders to be excluded by timing synthesis. The tool first enumerates upper and lower bounds immediately derivable from the given time bound information. The computational complexity of this enumeration process grows only linearly with respect to the number of parameters (we need to do an enumeration for each parameter, and enumerations for different parameters are independent of each other). The tool then searches over all possible covering upper bound sets and distributed lower bound sets. When the tool finds a matching pair of a covering upper bound and a distributed lower bound set, it derives timing constraints in the same way as demonstrated in Examples 5 and 6.

The current prototype assumes both lower and upper bounds (p_i and P_i, respectively) are defined for all pairs with bounds $(\pi_i, \Pi_i) \in actions(A)_\perp \times \mathcal{P}(actions(A))$.[5] Therefore, the underlying TIA has the lower bound parameter set $\{p_i\}_{i=1}^n$ and the upper bound parameter set $\{P_i\}_{i=1}^n$, both of which contain the same number of timing parameters, and a lower bound is at most as large as the matching upper bound: $p_i \leq P_i$.

A *linear term over lower bound parameters* $\{p_i\}_{i=1}^n$ is in the form $c_1p_1+c_2p_2+\cdots+c_np_n$, which we also write as $\sum_{i=1}^n c_ip_i$, where c_i is an integer constant for $1 \leq i \leq n$. A *linear term over upper bound parameters* $\{P_i\}_{i=1}^n$ is defined analogously.

An inequality the tool derives from one pair of a covering upper bound set and a distributed lower bound set has the form $\phi > \psi$, where $\phi = \sum_{i=1}^n c_ip_i$ is a linear term over lower bound parameters and $\psi = \sum_{i=1}^n d_iP_i$ is a linear term over upper bound parameters. The tool in general finds in a given event order multiple matching pairs of covering upper bound sets and distributed lower bound sets, for each of which it can derive a linear inequality. In such a case, multiple inequalities can be derived, and the given event order appears in no system execution if at least one of the inequalities is satisfied. Thus, the tool derives a *disjunction of linear inequalities* for one given event order.

The user typically needs to exclude multiple bad event orders. All specified event orders can be excluded if all disjunctions of linear inequalities derived from the event orders are satisfied. Therefore, a timing constraint derived by the tool forms a *conjunction of disjunctions of linear inequalities* – in a form similar to conjunctive normal form of Boolean logic, but in our case we have linear inequalities instead of Boolean variables: $\bigwedge_{i\in I} \bigvee_{j\in J_i} L_{i,j}$, where $L_{i,j}$ is a linear inequality.

The constraint derived by the tool may first contain some unrealizable inequalities (for example, an upper bound for a specific action set is strictly smaller than a lower bound for the same action set), or redundant inequalities (for example, one inequality is weaker than or equivalent to another inequality in a disjunction). We use a simple simplification algorithm to prune these inequalities. The details of this simplification algorithm is described in [31].[6]

Scalability experiment: To obtain a rough idea of the scalability of the constraint derivation process of the prototype with respect to the event order length, we conducted an experiment on deriving a constraint for randomly generated event orders of the train-gate example. This experiment (and all other experiments in this paper) was conducted on a desktop computer with an Intel Core$^{\mathrm{TM}}$2 Quad at 2.66 GHz and 4GB memory. We experimented with ten randomly generated event orders with length of thirteen, and the tool finished the constraint derivation pro-

[5] After obtaining a constraint simplified by the tool, the user can manually substitute $p_i = 0$ for (π_i, Π_i) with only an upper bound, and can substitute $P_i = \infty$ for (π_i, Π_i) with only a lower bound. The current prototype does not make use of this information of "unbounded in one side" in a simplification of a constraint, and this is our future work.

[6] Note that this simplification process is completely independent of constraint derivation process, and is provided by the tool for user's convenience. The user could instead manually simplify the derived constraint or could use external linear-logic simplification tools as well. This is different from the timed/hybrid model-checkers like HyTech, RED, TRex, and LPMC which inherently need an intelligent linear-logic simplification scheme to conduct a fixed-point calculation for reachable states symbolically expressed by a linear logic expression.

cess within one second for all experiments. Considering that the length of the event orders that we identified for the case studies presented in Section 7 are all less than ten, the results of these experiments are satisfactory. However, we have to conduct more case studies in order to examine the order of the length of the bad event orders in larger real-time systems.

Discussion: Though the current prototype does not treat a "disjunctive" language construct (such as $\cup$ of a regular expression), it is easy to derive a constraint for an event order that uses such a construct at the top level. For example, suppose we want to exclude a (pseudo) event order $e_1 e_2 \{e_3^1, e_3^1\} e_4$, which specifies that the third event order is either e_3^1 or e_3^2. We can simply treat this event order as two distinguished event orders $e_1 e_2 e_3^1 e_4$ and $e_1 e_2 e_3^2 e_4$.

Similarly, to exclude an execution that matches both of two event orders E_1 and E_2 ($E_1 \cap E_2$ in a regular expression), we can individually derive constraints for E_1 and E_2, and then disjunct them to obtain a constraint (at least one of $E1$ and E_2 needs to be excluded to exclude $E_1 \cap E_2$). Since we disjunct disjunctions of linear inequalities derived for E_1 and E_2, the derived constraint for $E_1 \cap E_2$ is a disjunction of linear inequalities. Thus, derivation of a constraint from $E_1 \cap E_2$ (among other ordinary event orders) does not destruct the conjunction-of-disjunctions structure of the final constraint.

7 Case Studies

7.1 Train-Gate Problem

In this section, we illustrate the user of EOA and the prototype tool using the train-gate example Train||Gate||SM that we have used in earlier sections of the paper.

We identified the following ten event orders to exclude all bad executions in the same way as described in Section 4.

A_1. $\perp$-Pass : insert {Check(false)} to $[0, 1]$
A_2. $\perp$-Request-Pass : insert {Check(false)} to $[0, 1]$
A_3. $\perp$-Request-Check(true)-Pass : insert {Check(false)} to $[0, 1]$
A_4. Pass-Open-Pass : insert {Check(false)} to $[2, 3]$
A_5. Pass-Open-Request-Pass : insert {Check(false)} to $[2, 3]$
A_6. Pass-Open-Request-Check(true)-Pass : insert {Check(false)} to $[2, 3]$
A_7. Pass-Pass
B_1. $\perp$-Request-Check(true)-Close-Open-Pass : insert {Check(false)} to $[0, 1]$, {Check(false)} to $[4, 5]$
B_2. Pass-Open-Request-Check(true)-Close-Open-Pass : insert {Check(false)} to $[2, 3]$, {Check(false)} to $[6, 7]$
C_1. Close-Pass-Request

We can classify these event orders into three groups. The first group (A_1 - A_7) represents a situation that the train passes the crossing before the gate becomes closed. A_1, A_2, and A_2 are the event orders used as a first candidate set of bad event orders in Section 4. In A_4, A_5, and A_6, the $\perp$ symbol in A_1, A_2, and A_3, respectively, is replaced by Pass-Open, so that they specify situations similar to A_1, A_2, and A_3, but after at least one Pass events have been performed. A_7 is like A_4, but without Open after Pass. The second group (B_1 and B_2) represents a situation that the gate becomes open too fast after it becomes closed, and thus the gate is open when the train passes the crossing. B_1 and B_2 intrinsically represents the same situation, but the $\perp$ symbol in B_1 is replaced by Pass-Open, so that B_2 specifies a situation after at least one Pass events have been performed. The third group (C_1) represents a situation that the gate becomes open again too late, that is, after the train makes a next request. Since all state variables of the gate automaton are reset when the gate becomes open (by Open event), if the gate becomes open after a request from the train, the request information is reset, and thus the gate would not become closed.

In this bad order identification process, we manually constructed a monitor (a classical finite state machine) for each of the identified ten event orders. Each monitor raises a flag exclude when it finds a subsequence of actions that match the underlying event order in a current automaton execution. Actually, we could (manually) combine some of the monitors[7] and needed to construct only six monitors (EOM1 - EOM6) at last. Model-checking for each refinement step took less than one second. At the end of the bad order identification step, we successfully model-checked the property $\Box(\neg$bad_event_order$) \Rightarrow \Box(\neg$SM.propertyViolated$)$ for Train||Gate||SM||EOM1||$\cdots$||EOM6, using a SAL symbolic model-checker [9], where bad_event_order = EOM1.flag $\vee$ EOM2.flag $\vee \cdots \vee$ EOM6.flag.

For event orders A_2, A_3, A_5, and A_6, we had to do a "decomposition" of an event order. For example, we cannot directly derive a meaningful constraint from A_3. In A_3, unlike the event order E_1 depicted in Figure 3, we have possibly unbounded number of Check(false) events before Request, and these Check(false) events are ignored. Thus, the bounds corresponding to $(\Delta, [0, 1])$, $(\Delta, [1, 2])$, $(\Delta, [1, 3])$, $(\delta, [0, 1])$, and $(\delta, [1, 3])$ in Figure 3 are removed from the set of enumerated bounds for deriving a constraint for A_3, and therefore we cannot derive the same constraint as in Example 5. Decomposing A_2 into the following two event orders resolved this problem: one with no Check(false): $A_3' = \perp$-Request-Check(true)-Pass, and one with one or more Check(false) events: $A_3'' = \perp$-Check(false)-Request-Check(true)-Pass : insert {Check(false)} to $[0, 1]$. A_3'' still has an IES, but for this event order, an upper bound removed from the upper bound set of E_1 in Figure 3 is only $(\Delta, [0, 1])$, and thus we can derive the same constraint $p > R + T + \Delta$ as in Example 5. We decomposed A_3, A_5, and A_6 similarly to the case of A_2 described above. We manually decomposed event orders for this case study, and automation of this decomposition is future work. More detailed analysis and automation of this decomposition process is our future work.

After the decompositions of A_2, A_3, A_5, and A_6, we had fourteen event orders, and the tool derived the following set of constraints from the given event orders after automatic simplification (the total time of derivation and simplification took less than

[7] For example, by changing the initial state of the monitor for A_4, we could also treat A_1. Same for the pairs of A_2 and A_5; A_3 and A_6; and B_1 and B_2.

one second): 1. $(p > R+T+\Delta)$; 2. $(r+t+c > P \vee \delta+t+c > P)$; and 3. $(r > C)$. The tool indicated that the first constraint was originally derived from a decomposed A_6, the second from B_1, and the third from C_1. Therefore, we obtained a constraint for each of the three groups we explained above.

7.2 Biphase Mark Protocol

A biphase mark protocol [25] is a lower-layer communication protocol for consumer electronics. Several researchers have conducted formal verification of this protocol (for example, [25, 32]), but as far as we know, completely automatic verification of it has not been done. We identified 22 bad event orders. (We successfully model-checked the discretized model under the condition that 22 bad event orders do not occur, and model-checking for each refinement step took less than one second). This number may look large, but similarly to the train-gate example in Section 7.1, we identified multiple event orders from a single bad situation (there were eight bad situations). Eight event orders (derived from three situations) had to be decomposed as in the case of the train-gate example. The tool derived three constraints (it took less than one second), which are equivalent to the three conditions manually derived in [32]. We actually had two more additional constraints for the first timing parameter synthesis attempt. One additional constraint was derived due to one event order that actually need not be excluded. The other constraint was derived because for one particular bad scenario, we supplied a shorter event sequence than needed to derive a weaker constraint (which is equivalent to one of the three constraints in [32] that we also derived in the first attempt.) By removing the first event order, and adding more event sequence to the second one, in the second attempt, we succeeded to derive three constraints as manually derived in [32].[8] A more detailed report on this case study will appear in a forthcoming publication [29].

7.3 IEEE 1394 Root Contention Protocol

The IEEE 1394 standard specifies communication infrastructure between electric devices. By using IEEE 1394, up to 63 devices can be connected in a tree topology. The root contention protocol (RCP) that we studied is used at the last phase of the tree topology identification. Though the bad scenarios to be excluded are two, due to the interleaved process actions (events), we ended up having 42 event orders. The model-checking successfully completed under the ordering assumption within one second. The tool derived a set of constraints that are equivalent to those manually derived in [27]. A more detailed report on this case study will appear in a forthcoming publication [29].

[8] In the conference version of this paper [30], we (from a misunderstanding) mentioned that the authors of [32] have used an additional constraint other than the derived three, in order to complete the proof. However, this turned out to be not the case: we contacted with Frits Vaandrager, one of the authors of [32], and he kindly clarified that the additional property of the system mentioned in [32] was not additionally assumed, but was actually just discovered as an invariant of the system.

7.4 Fischer Mutual Exclusion

The Fischer mutual exclusion algorithm ([21], Section 24.2) is a mutual exclusion algorithm that uses a timing behavior for correctness. We identified one bad event order, by using the symmetry among process behavior. In this event order, we focus on a specific interleaving of events between a pair of processes. Ignored event specifications are used to treat behavior of other processes than the focused pair as “don’t-care”. We successfully model-checked the discrete model under the correct ordering assumption (it took 40 seconds for a system with five processes). The tool derived the constraint that is manually derived in [21].

8 Conclusion and Future Work

In this paper, we presented the *event order abstraction* (EOA) technique to parametrically verify real-time systems. By using EOA, the user can directly make use of his/her intuition about what kind of bad scenarios need to be prevented, by specifying bad event orders. We demonstrated the applicability of the technique by a simple train-gate system and a summary of three other case studies, a biphase mark protocol, the root contention protocol of IEEE 1394, and the Fisher mutual exclusion algorithm, are briefly reported.

This technique can be extended by enhancing automation of verification using EOA in the following processes: construction of an event order monitor, decomposition of an event order, and extraction of a bad event order using heuristics. An interesting future direction is extending bad event order language to treat a partial order of events, as well as the current sequential order.

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. Along this line, identified bad event orders could be used in model-based testing or model-based test-case generation [10, 8], in which a formally specified model is used to test an actual implementation of a system.

Acknowledgment: First of all, I thank my supervisor, Prof. Nancy Lynch, for her patient guidance on this research and fruitful comments on an earlier version of the paper. Also, several comments from Eunsuk Kang helped me revise the paper. I also thank anonymous reviewers for their helpful comments.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *ACM Symposium on Theory of Computing*, pages 592–601, 1993.

[3] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A tool for reachability analysis of complex systems. In *Computer Aided Verification*, pages 368–372, 2001.

[4] E. Asarin, O. Maler, and A. Pnueli. On discretization of delays in timed automata and digital circuits. In *Proc. of CONCUR'98*, volume 1466 of *Lecture Notes in Computer Science*, pages 470–484, Nice, France, 1998. Springer.

[5] D. Bosnacki. Digitization of timed automata. In *Proc. of FMICS 99*, 1999.

[6] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip-synchronisation protocol using uppaal. *Formal Aspects of Computing*, 10(5-6):550–575, 1998.

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[8] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *International Conference on Software Engineering*, pages 285–294, 1999.

[9] L. M. de Moura, S. Owre, H. Rueß, J. M. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *Proc. of CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer, 2004.

[10] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993. Springer-Verlag.

[11] G. Frehse, S. K. Jha, and B. H. Krogh. A counterexample-guided approach to parameter synthesis for linear hybrid automata. In *HSCC 2008*, volume 4981 of *Lecture Notes in Computer Science*, pages 187–200. Springer, 2008.

[12] K. Havelund, A. Skou, K. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using uppaal. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 2, Washington, DC, USA, 1997. IEEE Computer Society.

[13] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. Technical Report MIT/LCS/TM-511, MIT, 1994.

[14] T. Henzinger, J. Preussig, and H. Wong-Toi. Some lessons from the HYTECH experience. In *Proc. of the 40th Annual Conference on Decision and Control*, pages 2887–2892. IEEE Computer Society Press, 2001.

[15] T. A. Henzinger. The theory of hybrid automata. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 278, Washington, DC, USA, 1996. IEEE Computer Society.

[16] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *REX workshop Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 226–251. Springer-Verlag, 1992.

[17] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proc. of ICALP 1992*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer, 1992.

[18] T. Hune, J. Romijn, M. Stoelinga, and F. W. Vaandrager. Linear parametric model checking of timed automata. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 189–203, 2001.

[19] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2006.

[20] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[21] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[22] O. Maler and S. Yovine. Hardware timing verification using kronos. *iccsse*, 00:23, 1996.

[23] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurent Systems: Specification*. Springer-Verlag, 1993.

[24] M. Merritt, F. Modugno, and M. R. Tuttle. Time-constrained automata (extended abstract). In *Proc. of CONCUR 1991*, volume 527 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1991.

[25] 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.

[26] J. Ouaknine and J. Worrell. Revisiting digitization, robustness, and decidability for timed automata. In *Proc. of the 18th IEEE Symposium on Logic in Computer Science (LICS'03)*, pages 198–207, 2003.

[27] D. P. L. Simons and M. Stoelinga. Mechanical verification of the IEEE 1394a root contention protocol using Uppaal2k. *International Journal on Software Tools for Technology Transfer*, 3(4):469–485, 2001.

[28] R. Spelberg and W. Toetenel. Parametric real-time model checking using splitting trees. *Nordic Journal of Computing*, 8:88–120, 2001.

[29] S. Umeno. Parametrically verifying embedded real-time protocols using event order abstraction. Technical report, Massachusetts Institute of Technology. To appear.

[30] S. Umeno. Event order abstraction for parametric real-time system verification. In *EMSOFT 2008: International Conference on Embedded Software*, pages 1–10, October 2008.

[31] S. Umeno. Event order abstraction for parametric real-time system verification. Technical Report MIT-CSAIL-TR-2008-048, Massachusetts Institute of Technology, July 2008.

[32] F. W. Vaandrager and A. de Groot. Analysis of a biphase mark protocol with UPPAAL and PVS. *Formal Asp. Comput.*, 18(4):433–458, 2006.

[33] F. Wang. Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures. *Transactions on Software Engineering*, 31:38–51, 2005.

[34] S. Yovine. KRONOS: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2):123–133, 1997.

[35] D. Zhang and R. Cleaveland. Fast on-the-fly parametric real-time model checking. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 157–166, 2005.