Event Order Abstraction for Parametric Timed 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 preserved by timing dependent behavior of the systems. By using EOA, one can separate the task of verifying a system into two parts: 1. A derivation of timing parameter constraints for correct orderings of events in the system; and 2. A safety property verification of the system given that those correct orderings are preserved. We first identify bad event orderings. Then we automatically derive a set of timing constraints under which the system does not exhibit the identified bad event orderings. In parallel to a derivation of timing constraints, by using an ordinary untimed model-checking, we examine whether a discretized system model in which all timing behaviors are abstracted away satisfies a desirable safety property under the assumption that the identified bad event orders occur in no system execution. We successively refine this assumption by extracting additional bad event orders from a counterexample obtained from a model-checking. When we successfully model-check the discretized model under an ordering assumption and derive timing constraints for that ordering assumption, we have obtained a sufficient set of timing constraints under which the system executes correctly with respect to a given safety property. We summarize three case studies, a train-gate system, a biphase mark protocol, and the Fischer mutual exclusion algorithm.

1 Introduction

There are real-time systems whose timing dependent behavior is used to ensure "correct orderings of events" that appear in its execution (for example, a biphase mark protocol [10], the Fischer mutual exclusion algorithm ([8], Section 24.2), and the IEEE 1394 root contention protocol [11]). For these systems, certain timing constraints must hold for correctness. To verify the correctness of these systems, a verification engineer or researcher typically follows one of the following two approaches: 1. (*fixed-parameter verification*) By fixing the timing parameters (such as delays or time deadlines) in the system, reduce the system model to a more tractable one such as an Alur-Dill timed automaton [1] and model-check the reduced system (using UPPAAL [7] for instance); or 2. (*parametric verification*) Treat the timing parameters as uninterpreted constants, find an appropriate set of constraints for the timing parameters, and prove or mechanically check the correctness under the constraints [10, 14, 16]. The first approach is usually less expensive than the second one with respect to both the computational cost and the amount of human interaction. However, the results obtained from the second approach give us more information about the allowed parameter sets, and may give an engineer more freedom of parameter choices.

In this paper, we present a new abstraction technique, *event order abstraction* (EOA), for parametric safety verification of the above described class of real-time systems in which correct orderings of events are critical for correctness. By using EOA, one can separate the task of verifying a system into two parts: 1. A derivation of timing parameter constrains for correct orderings of events in the system; and 2. A safety property verification of the system given that those correct orderings are preserved. A parametric verification of a real-time system using EOA is conducted in the following steps. First, we identify "bad" event orders that we want to exclude from the system executions. This can be done by simulating the underlying system, or by conducting a model-checking for a "discretized" model of the system in which all timing constraints are removed, and obtaining a counterexample. We express bad event orders by a simple language that can express an order of events and some types of repetition of events. Next, by a scheme that we will present in the paper, we automatically derive a set of timing constraints under which bad event orders are not exhibited. In parallel to a timing constraint derivation, we check whether or not the discretized model of the system satisfies a desirable safety property under the condition that the model does not exhibits the specified bad event orders. We conduct this check by constructing a monitor[1] that raise a flag when one of the specified bad event orders is exhibited, and model-checking the discretized model with this monitor under the condition that

*To be presented at the Second Workshop on Event-Based Semantics, April 21st, 2008. St. Louis. USA. The paper first appears online on April 4th, 2008. The full version of the paper has been submitted for publication.

[1] We manually constructed monitors for the case studies presented in the paper (since the construction was straightforward for these case studies), but we are planning to develop a monitor construction tool to enhance automation of the verification process.

the monitor does not raises a flag in an underlying execution (in Linear Temporal Logic (LTL) [9], this condition can be represented by $(\Box(\neg Monitor.flag)) \Rightarrow (\Box(\neg DiscretizedModel.bad)))$). Lastly, if the model-checking is completed with a positive answer, we have obtained a set of timing constraints under which the system satisfies the given safety property. If we obtain a counterexample, then we extract another bad event order from it, and repeat the same process until we successfully model-check the discretized model. We extract a bad event order manually, since we need some human insights to do so.

Related works: Some of the existing model-checkers for real-time systems (HYTECH [5], RED [15], TReX [2], LPMC [12], and an extension of UPPAAL [6]) allow *automatic synthesis* of timing parameters for a given desirable property: these tools automatically derive constraints on timing parameters for the system to satisfy a given property. The main differences of EOA from the existing automatic constraint derivation tools listed above are the following three: First, to use EOA, the user has to provide a set of desirable ordering of events to be excluded in the system. Second, EOA can treat a class of systems that may exhibit an unbounded number of repetitions of events. Third, when doing successive refinements by using EOA, each abstracted model constructed in these refinements is a completely discrete transition system. The existing parametric model-checkers listed above use a forward and backward reachability analysis for a model with symbolically represented states. 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 [5], 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). The third difference implies that by using EOA, the verifier can directly employ existing verification techniques for a discrete transition system. In addition, since the model-checking process of EOA does not have to manipulate linear inequalities directly like the existing parametric timed system model-checkers, EOA does not suffer from the "dimensionality" problem: an automatic synthesis of these model-checkers rapidly becomes intractable as the number of parameter grows ([5], Section 5. Lessons learned).

The rest of the paper is organized as follows. In Section 2, we introduce a new automaton framework, *time-interval automata*. The framework is an extension of the I/O automata framework [8], and with this framework, one can specify lower and upper time bounds between one action and its succeeding actions. In Section 3, we explain how we specify event orders. Section 4 is devoted to presenting a scheme of an automatic timing constraint derivation. Section 5 presents case studies of parametric verification using EOA. We summarize three case studies, a train-gate system, a biphase mark protocol that has been studied in several verification papers (for example, [10, 14]) and the Fischer mutual exclusion algorithm ([8], Section24.2). We conclude in Section 6.

2 Time-Interval Automata

The *time-interval automata* (TIA) framework is an extension of the I/O automata (IOA) framework [8]. 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 for an underlying I/O automaton. A time bound for action a and actions in B is represented as an interval in the form $[l, u]$. This bound represents that, for any time of occurrence t_a of action a, no action in B occurs before $l + t_a$, and at least one action in B is performed before or at $u + t_a$. When we define such a time bound $[l, u]$ between an action a and set of actions B, we require that at least one action in B be enabled after a is performed, and continue being enabled until b is performed. In this way, we make sure that b is indeed enabled in interval $[l + t_a, u + t_a]$. An *interval-bound map* specifies this time bound. 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. An interval-bound map by itself may not satisfy requirements to express a meaningful bound (for example, the specified lower bound is no greater than the specified upper bound). We say that an interval-bound map is *valid* if it satisfies the requirements (formal definitions for the TIA framework appear in the full version of this paper [13]).

Definition 1 (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 2 (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 (a formal definition of these requirements appear in [13]).

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).

Example 1 (Time-Interval Automaton). We describe an example of time-interval automata. The example is inspired from railroad crossing problems [4]. The example is constructed from a composition of a train automaton (Figure 1) and a gate automaton (Figure 2). An informal

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 $\neg$requested
eff requested := true;
output Pass
eff requested := false;

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

Figure 1: Train automaton

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 that the train passes the crossing, so that cars can cross the railroad. When the train gets close to the crossing, it *requests* to close the gate. 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 open, it becomes closed after a bounded time interval.

The actions of the Train automaton models actions taken by the train in the railroad. The Request action represents an close request made by the train to the gate. The Pass action represents that the train passes the crossing. The automaton has four bounds for these two actions. The first one ($b(\perp, \{\text{Request}\}) = [r, R]$) 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 ($b(\perp, \{\text{Pass}\}) = [p, P]$) and the forth 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.[2] 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 an 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

[2]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 $P = L/v_{min}$.

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 $\wedge$ open
eff open := false;
output Open
pre $\neg$open
eff open := true;
train_requested := false;
check_succeeded := false;
output Check(result)
pre $\neg$check_succeeded $\wedge$ result = train_requested
eff check_succeeded := train_requested;

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

Figure 2: Gate automaton

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) within 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 monitor automaton Monitor that monitors output actions Pass, Close, and Open from Train and Gate, and set its state variable bad to true if Pass occurs when the gate is open. The invariant (safety property) we want to check is: for any reachable state of Train||Gate||Monitor, Monitor.bad = 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.

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) shown in Example 1 matches any execution of (Train||Gate) that contains a Request ac-

tion immediately followed by a Pass action. An event order may start with a $\perp$ symbol, which specifies that the event order matches with 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.

Example 2 (Event order). An example of event orders that we want to exclude in Train||Gate||Monitor discussed in 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 to detect it, and starts closing the gate. However, the gate close request is detected too late relative to the speed of closing 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 5), some event orders to be excluded cannot be represented in a form of a simple event order like the ones we consider earlier in this section. 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, the number of possible failing checks that precede the Pass event is unbounded when the relation between δ, Δ, r, and R is unknown. 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 any occurrence 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 appears in the full version of the paper [13].

A formal definition of a match between a timed execution of a TIA and an event order (possibly with an IES) appears in [13]. We refer to an execution that matches with E as *E-matching execution*.

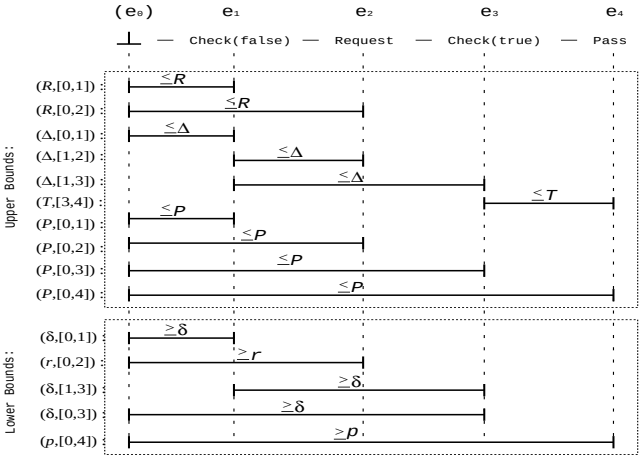

Figure 3: Upper and lower bounds for the event order E_1

4 Deriving Timing Constraints

In this section, we present a scheme to derive a set of timing constraints to exclude an execution that matches with a given event order.

Given an event order E and a bound map b of a TIA, we first derive the upper and lower bounds between the time of occurrence of two events in E. from the upper and lower bound conditions for a timed execution of a TIA. They are tagged with the event-index interval for which they are derived (the $\perp$ symbol is treated as the zero-th event e_0).. An upper bound for an event-index interval $[i,j]$ is constructed from the fact that a particular event *does not appear* in $[i,j]$, whereas a lower bound for $[i,j]$ is constructed from the fact that particular events appear at i and j. An upper bound u and a lower bound l is constructed in a way that, for any execution $\alpha = s_0(\pi_1,t_1)s_1\cdots$ that matches with $E = e_1e_2\cdots e_n$ (possibly with an IES), the matched subsequence of actions $\beta = \beta_0\pi_{k_1}\beta_1\cdots\beta_{k_{n-1}}\pi_{k_n}$ (where β_i's are the ignored part, and π_{k_i} matches e_i) satisfies $t_{k_j} - t_{k_i} \leq u$, and $t_{k_j} - t_{k_i} \geq l$.

Example 3 (Upper and lower bound sets). We show an example of construction of upper and lower bounds. The underlying automaton is Train||Gate2||Monitor 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 and lower bounds for E_1.

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 $upper(\perp,\{\mathsf{Request}\}) = R$ 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).

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 $lower(\mathsf{Check(false)},\{\mathsf{Check(false)},\mathsf{Check(true)}\})) = \delta$

defined in the bound map.

We need a notion of a covering upper bound set and a distributed lower bound set to combine individual bounds and synthesize a meaningful timing constraint. Informally, a covering upper bound set U for an event interval I is a set of upper bounds such that when we take a union of all intervals for which upper bounds in U are derived, the union becomes I (intervals corresponding to upper bounds in U *cover* I). A distributed lower bound set L for an event interval I is a set of lower bounds such that each interval for which a lower bound in L is constructed is contained in I, and all intervals for which lower bounds in L are constructed do not overlap (intervals corresponding to lower bounds in L are *distributed* in I, without overlapping). A formal definition appears in [13].

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.

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).

Theorem 1 *Consider an event order E. A time-interval automaton (A, b) exhibits no execution that matches with E if there exists a set of upper bounds $U = \{u_m\}_{m=1}^{p}$, a set of lower bounds $L = \{l_r\}_{r=1}^{q}$, 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 || Gate2) 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. We have an upper bound $upper(\perp, \{\mathsf{Request}\}) = R$ defined for Train||Gate2, 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 first Check event must occur before Pass.

Implementation: We implemented a prototype of a timing constraint derivation algorithm. The algorithm searches over all possible covering upper bound sets and distributed lower bound sets, and derives timing constraints in the same way as demonstrated in Example 5. A derived constraint set may contain some redundant constraints (for example, one constraint is weaker than or equivalent to another constraint) or unrealizable constraints (for example, an upper bound for a specific action set is strictly smaller than a lower bound for the same action set). We use a simple simplification algorithm to prune these constraints. We usually need to exclude multiple event orders, not just one. The prototype tool can also manage constraints derived for multiple event orders, and does simplification over these constraints.

5 Case Studies

5.1 Train-Gate Problem

In this section, we summarize a case study of EOA for the train-gate example Train||Gate||Monitor that we have used in earlier sections of the paper.

We identified the ten event orders to exclude all bad executions. We can classify these event orders into three groups. The first group (consisting of seven event orders) represents a situation that the train passes the crossing before the gate becomes closed The second group (consisting of two event orders) represents a situation that the gate becomes open too fast after it become closed, and thus the gate is open when the train passes the crossing. The third group (consisting of one event order) represents a situation that the gate becomes open too late – after the train makes a next request. Since all state variables of the gate automaton are reset when the gate becomes open again, if the gate becomes open after a request from the train, the request information is reset, and thus the gate will not be closed.

The tool derived the following set of constraints after an automatic simplification (the derivation took less than 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 is originally derived from an event order in the first group, the second constraint

from an event order in the second group, and the third constraint from an event order in the third group. Therefore, we obtained a constraint for each of the three groups we explained above.

We constructed monitors (classical finite state machines) for the identified ten event orders. Each monitor raises a flag exclude when it founds a subsequence of actions that match the underlying event order in a current automaton execution. We successfully model-checked the property $(\Box(\neg \mathsf{bad\_event\_order})) \Rightarrow (\Box(\neg \mathsf{Monitor.bad}))$ for the train-gate model with these event order monitors using a SAL symbolic model-checker [3]. The model-checking time was less than one second.

5.2 Biphase Mark Protocol

A biphase mark protocol [10] is a lower-layer communication protocol for consumer electronics. Several researchers have conducted formal verification of this protocol (for example, [10, 14]), but as far as we know, completely automatic verification of it has not been done. We identified 22 bad event orders. This number may look large, but similarly to the train-gate example in Section 5.1, we identified multiple event orders from a single bad situation (there were six bad situations). The tool derived five constraints (it took less than one second). Three of them are equivalent to the three conditions manually derived in [14], and one is automatically satisfied under the protocol model in [14]. The remaining constraint is not reported in [14], but we believe that the constraint must hold for correctness (it is needed to exclude a simple bad scenario). We successfully model-checked the discretized model under the condition that 22 bad event orders do not occur (it took less than one second).

5.3 Fischer Mutual Exclusion

The Fischer mutual exclusion algorithm ([8], 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". The tool derived the constraint that is manually derived in [8]. We successfully model-checked the discrete model under the correct ordering condition (it took 40 seconds for a system with five processes).

6 Conclusion and Future Work

In this paper, we presented *event order abstraction* (EOA) technique to parametrically verify real-time systems. Applicability of the technique is demonstrated by three case studies, a train-gate system, a biphase mark protocol, and the Fisher mutual exclusion algorithm. We are planning to enhance 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. We have started analyzing the IEEE 1394 root contention protocol [11] as another case study of EOA. The result will appear in the future publication.

Acknowledgment: I thank Prof. Nancy Lynch for her patient guidance on this research and helpful comments on an earlier version of the paper.

References

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

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

[3] 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.

[4] 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, 1994.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

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

[13] S. Umeno. Event order abstraction for parametric real-time system verification. Technical report, Massachusetts Institute of Technology. To appear.

[14] 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.

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

[16] 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.