Timeout Order Abstraction for Time-Parametric Verification of Loosely Synchronized Real-Time Distributed Systems

Shinya Umeno and Nancy Lynch
CSAIL, Massachusetts Institute of Technology, Cambridge MA, USA
{umeno,lynch}@csail.mit.edu

***Abstract*— We present *timeout order abstraction (TO-abstraction)*, a technique to systematically abstract a given loosely synchronized real-time distributed system (LSRTDS) into an untimed model. We define the subclass of LSRTDS's that we can apply TO-abstraction using a syntax template that represents a restriction to Tempo, the primary modeling language of TIOA [7]. The untimed model obtained from the abstraction is a classical finite state machine, and thus one can automatically verify temporal properties of the model using a conventional model-checker. We prove the soundness of the abstraction using simulation relation. From this result, we guarantee that any untimed safety property of the untimed model also holds for the original TIOA model.**

We have applied TO-abstraction to a resource-sharing protocol and the DHCP Failover protocol. We verified untimed abstractions of them by bounded model-checking up to depth 20. We have also experimented with effectiveness of bug-finding using our technique by mutating particular parts of the original code. From this experiment, we found a complex bad execution that would have been very difficult to find by human or simulations.[1]

1. Introduction

Analyzing correctness of real-time distributed systems is a challenging problem due to the combination of nondeterminism from process interleaving and timing constraints in the system executions. The class of *Loosely-synchronized real-time distributed systems* (LSRTDS's) is an interesting subclass of real time distributed systems. In this subclass, the processes or modules in the system are assumed to have *loose synchronization*, that is, there is an a priori known upper bound ε on the skew between local clocks in processes. Processes communicate *time data* (timing-related information such as time stamps) with each other, and set their *timeouts* using time data. These timeouts are used to constrain processes' behavior in such a way that the processes execute a certain designated action before or after other processes execute another designated action. For example, the DHCP Failover protocol [5], used for one of the two case studies in this paper, falls into this subclass. Due to a general assumption about evolution of local clock values and timeout setting using time data communication between processes, automatic exhaustive exploration (model-checking) techniques for LSRTDS's has not been studied thus far in the community, as far as we know. In particular, existing timed and hybrid model-checkers cannot directly treat the subclass of LSRTDS's that we treat in this paper. We tackle in this paper the verification problem for LSRTDS's by providing a way of machine-assisted automatic analysis for a particular subclass of LSRTDS's.

We present *timeout order abstraction (TO-abstraction)*, a technique to systematically abstract an LSRTDS into an untimed model. We define the subclass of LSRTDS's that we can apply TO-abstraction using a syntax template that represents a restriction to Tempo, the primary modeling language of TIOA [7]. The TIOA framework has been used to model and verify (with hand proofs) several real-time distributed systems and algorithms (for example, [8], [9], [3], [4], [5], [14]). TO-abstraction enables the user to conduct *time-parametric verification* of a given TIOA model described by the template in the sense that the local clock skew bound ε and the special timing-related constant that we explain later are treated as parameters of the system, and therefore, are not instantiated into concrete values. TO-abstraction performs a code-to-code conversion by which a model described using the template is converted into ordinary (untimed) I/O automaton code. The untimed model obtained by TO-abstraction is a finite-state machine, and thus one can automatically verify (untimed) temporal properties of the model using a conventional model-checker. We prove a simulation relation from the original TIOA model to the abstraction using the loose synchronization assumption described earlier. From this result, we guarantee that any untimed safety property of the untimed model also holds for the original TIOA model. This soundness theorem works as a *meta theorem* for the template: the theorem applies to every system described by the template.

The template can express interesting building blocks that processes in the system can use to communicate time data and set timeouts. For instance, a process can pick a *time nonce*, an arbitrary value that represents some future time on its local clock. The process sends the time nonce to other processes. By intelligently setting timeouts using the time nonce, one group of processes can time out after another group of processes have done so. A process may update its timeout time or variables using a "max" operation for pieces of time data. This max operation is typically used to conservatively update the estimate of other processes' timeout times. A process may check whether or not time data stored in their state variables or time data received from other processes has already "expired" (its local clock value already exceeds the value of the time data).

Contributions: There are three main contributions in the presented work.

First, we provide an abstraction technique, timeout order abstraction (TO-abstraction), for formal verification of a specific subclass of LSRTDS's, which have not been studied in the context of automated formal verification. As far as we know, TO-abstraction presented in this paper is the first technique that enables the user to reduce a verification problem of a LSRTDS to a finite-state model-checking problem. By conducting case studies, we have found that TO-abstraction is useful not just for verification (developing guarantees), but also for bug-finding. We have experimented with "mutating" parts of the code for a case study of the technique, and we found, by model-checking the abstracted model, a counterexample that is complex and thus is arguably difficult to find by human or simulations.

This research has been supported by the NSF Award CCF-0702670, the NSF Award CNS-0614414, and the AFOSR/DOD CAP Funds.

[1]A supplemental information about this paper (such as more detailed explanation of parts of the paper or proofs), and the code used for the case study can be obtained from the URL cited as [11].

Second, the template presented in the paper provides protocol designers with interesting building blocks for real-time protocols. In particular, the *time-stamp-estimation trick* that we explain in Section 2 gives the designers an interesting way of adding fault-tolerance to their protocols. This special trick is used in [5] without particularly mentioning its usefulness, intuition, or subtlety.[2] We make this special usage explicit by having it as one of the building blocks that can be expressed by the template.

Third, we provide a case study on automatic verification of the DHCP Failover (DHCP-F) protocol. The protocol has been studied in [5] in the context of formal verification using manual (hand-written) proofs, but no study on automatic analysis of the protocol has been reported thus far. Our case study provides exhaustive exploration of scenarios of DHCP-F up to the execution length of 20 (20 discrete transitions, including sending and receiving messages, of the system) for the configuration of two clients and two servers.

Related work: There are several techniques that have been developed thus far for a reduction from real-time system verification to finite-state machine verification. The most famous technique is arguably the triangular-automaton-construction technique for timed automata, developed by Alur and Dill [1]. We have studied an abstraction technique, *event order abstraction*, for parametric timed verification by focusing on key event orders [10]. The class of LSRTDS's that we treat by TO-abstraction cannot be expressed by such frameworks as Alur-Dill Timed Automata [1], Linear Hybrid Automata [6], or Time-Interval Automata that we use in [10]. This is mainly due to the fact that the loosely synchronized assumption allows very general evolution of local clock values: as long as the local clock value evolves increasingly and the loosely synchronized assumption (the ε skew bound among the local clocks) are satisfied, the evolution can be arbitrary (and thus can be non-linear). Therefore, we cannot directly benefit from the existing verification techniques developed for these frameworks. We consider this fact as one of the main reasons that LSRTDS's have not been studied intensively in the automatic verification community.

The rest of this paper is organized as follows. Section 2 explains how we have found the template for TO-abstraction, and why the building blocks for time data communication expressed using the template are interesting. We discuss the usage of building blocks using small toy examples. In Section 3, we describe the settings of a distributed real-time system that we assume and the restricted syntax template for the TIOA programming language with which the modules in the system must be described in order to use TO-abstraction. Section 4 is devoted to presenting how we can abstract a TIOA model described using the template defined in Section 3 into a finite-state untimed model. In Section 5, we briefly explain how we can prove the soundness of TO-abstraction. In Section 6, we reports applications of TO-abstraction to a resource-sharing protocol and the DHCP Failover protocol. We conclude in Section 7.

2. Background

In this section, we explain how we found the template for TO-abstraction, and why their building blocks for time data communication are interesting. We discuss the usage of building blocks using small toy examples.

We started this research by analyzing the *DHCP Failover protocol (DHCP-F)*, and trying to find common patterns that satisfy both of the following properties: 1. The set of patterns is general, and other existing real-time protocols may have used it already, or a protocol designer can use it for a future design; and 2. Every protocol described by the set of patterns can be systematically abstracted into an untimed model that can be verified by a conventional model-checker. The *DHCP Failover protocol (DHCP-F)* [5] is an extension of the *Dynamic Host Configuration Protocol (DHCP)*, which is widely deployed for communication devices to automatically obtain an IP address on the Internet. A DHCP server offers a client an IP address in the form of a "lease" with an expiration time. DHCP-F supplements the ordinary DHCP with stronger fault tolerance using multiple backup servers – when the main server encounters a failure and becomes down, one of the backup servers takes over the main server's job. The main difficulty of using such backup servers is to maintain the consistent view of the lease periods of IP addresses across the main server and all backup servers. The time-stamp-estimation trick that we explain in Section 2 is used for this purpose.

We have found key building blocks of DHCP-F that other protocols can use under the loose-synchronization assumption. Processes in the protocol use the following two main ways of setting timeouts.

1) The first way of setting a timeout uses a *time nonce*, a value arbitrarily picked by a process.[3] A typical use of a time nonce is described in Protocol 1 in this section.
2) The second way uses a *time stamp*, plus a special fixed constant waiting time u. A time stamp is a value copied from the current value of the local clock of a process. Timeout setting using a time stamp and u can be considered a special form of setting timeout using a time nonce. This special form is used for processes to perform the *time-stamp-estimation* trick, used in Protocol 2.

In the rest of this section, we illustrate by examples how to use the building blocks in the template, and how a designer can add time data communication into his/her protocol in order to improve the system throughput and fault-tolerance.

We consider the following common setting for the examples. Two processes P_1 and P_2 share a resource, and they must not access the resource at the same time. Their strategy is time-sharing the resource by communicating with each other by sending messages through channels. Two processes' local clocks are loosely synchronized, and the skew between them is strictly less than ε. We assume that the values of their local clocks are monotonically increasing. We assume that for this time-sharing, P_1 first accesses the resource and then P_2 and P_1 alternately accesses the resource in turns. At first, we assume that the channel is stable and thus messages would not be lost and message contains would not become broken.

Without using timing-related information, the processes can use the following simple strategy: Each process sends a "done" message to the other process when it finishes its job. When a process receives a "done" message, it starts using the resource. This simple protocol is inefficient in some cases because one process cannot start the job until it receives the other process's

[2]The Internet-draft version of DHCP-F does not consider this type of subtle arguments about loosely synchronized clocks, and thus how long a leading server has to wait to conservatively estimate time stamps other servers have picked is not clearly stated. We contacted the first author of [5], and were informed that this subtle special usage, which we will call the time-stamp-estimation trick, was proposed by him, discussed by the authors of [5], and then adopted in the model of DHCP-F used in [5].

[3]A time nonce in an actual low-level implementation may be computed using, for example, a complex optimization and/or adaptive algorithm or a randomized algorithm. We just assume the most general (least restrictive) assumption: as long as its value is larger than the current clock of the process that picks the time nonce, it is a valid time nonce for our setting.

"done" message. If the message delay is relatively long compared to the resource usage time interval, inefficiency becomes a real problem. Processes can use time nonces to loosely synchronize their behaviors, as Protocol 1 shows. Protocol 1 makes more efficient use of the resource if the clock skew bound ε is smaller than the message delay.

Protocol 1: P_1 picks the time until which it will use the resource, as a time nonce TN_1, and sends it to P_2. P_1 sets a timeout at TN_1 and starts using the resource. When P_2 receives TN_1, it sets a timeout at $TN_1 + \varepsilon$. The skew bound ε is used to *most conservatively* estimate when P_1 times out, *on P_2's local clock*. When P_2 times out, it is guaranteed that P_1 has already timed out. Therefore, P_2 can *immediately* use the resource. P_2 picks another time nonce TN_2, sets a timeout at TN_2, and sends TN_2 to P_1. Upon receipt of TN_2, P_1 sets its timeout to $TN_2 + \varepsilon$. P_1 and P_2 repeat the same routine forever. □

Now we consider a different assumption for channels. Now the channels are not stable, and contents of messages sent between processes may become broken. When message contents become broken inside the channel, processes can recognize that the contents are broken (for example, using a check-sum). Under the above assumption, we cannot use Protocol 1 – if information of TN_1 sent from P_1 to P_2 becomes broken, there is no way P_2 can estimate when P_1 finishes its job. One (not so smart) option is going back to "done" message communications.

Using a time stamp with a constant waiting time instead of an arbitrarily time nonce resolves this situation. Suppose processes a priori share the value of the constant waiting time u, and the value of u is fixed.

Protocol 2: P_1 picks a time stamp TS_1, instead of an arbitrary time nonce, and sets its timeout to $TS_1 + u$. Therefore, P_1 uses the resource for u time units (measured on its local clock). P_1 sends $TS_1 + u$ to P_2. If the above message sent to P_2 is not broken, P_2 sets its timeout to $TS_1 + u + \varepsilon$ (the received time data plus ε) as in Protocol 1. If the message is broken, P_2 sets its timeout to the special value $clock_2 + u + 2\varepsilon$, where $clock_2$ is the current value of P_2's local clock. When P_2 times out, it immediately starts using the resource. It also picks a time stamp TS_2, sets its timeout to $TS_2 + u$, and sends $TS_2 + u$ to P_1. P_1 and P_2 repeat the same routine forever. □

Time-Stamp-Estimation Trick: The timeout setting using $clock_2+u+2\varepsilon$ in Protocol 2 is the special timeout setting, the *time-stamp-estimation* trick that we have mentioned in the introduction. We explain in the following why this value can be used to estimates conservatively when P_1 times out. We can interpret the value $clock_2 + u + 2\varepsilon$ as $(clock_2 + \varepsilon) + u + \varepsilon$. If $clock_2 + \varepsilon$ is equal to or greater than TS_1, then P_2 indeed succeeds in conservatively estimating P_1's timeout (if the message were not broken, P_2 would have set its timeout to $TS_1 + u + \varepsilon$). We consider in the following the moment that P_2's timeout is set to $clock_2 + u + 2\varepsilon$. From the loose synchronization assumption, the value of $clock_2 + \varepsilon$ is at least as large as the value of $clock_1$, the local clock of P_1. Because P_1's clock value is monotonically increasing, the current value of $clock_1$ is greater than the value of TS_1, which is copied from the past value of $clock_1$. Therefore, $clock_2 + \varepsilon \geq TS_1$, as needed. The key to the above argument was that because of the fact that P_2 received a (broken-content) message from P_1, P_2 was sure that P_1 had already picked time nonce TS_1. The value of $clock_2 + \varepsilon$ overestimates *any time stamp that has been picked thus far in physical time*, and a process can perform this estimation *by looking at just its local clock*.

We can extend time-sharing problem to two groups of processes, as we describe in the following Protocol 3.

Protocol 3: Two groups of processes Π_1 and Π_2 share a resource, and any two processes in the different groups must not use the resource at the same time. Processes in Π_1 first broadcast their "$TS + u$" values to Π_2, and processes in Π_2 wait until the *latest time* among received $TS + u$ values. Upon receipt of every $TS + u$ value, processes in Π_2 update an estimate of the latest time using a 'max' operation: $estimation := \max(received\_TS\_u, estimation)$, and updates the timeout time to $estimation + \varepsilon$. When a process receives a broken-content message, it uses the time-stamp-estimation trick, by setting a timeout at $\max(clock+u+2\varepsilon, old\_timeout\_time)$. When a process in Π_2 has received messages from all processes in Π_1 and has timed out, it can use the resource and broadcasts a $TS + u$ value to Π_1. Π_1 and Π_2 repeat the same routine forever. We can in addition give processes the choice of choosing their own "customized" resource-sharing time, instead of the constant time u: When broadcasting a resource-sharing time to another group, a process may choose to propose an arbitrary time using a time nonce, instead of $TS + u$. In this case, that process starts using the resource only after it gets acknowledgments from all processes in the different group (because the time-stamp-estimation trick cannot be used in this case). □

Protocol 3 above may exhibit subtle corner-case scenarios that are difficult to verify by human or simulations. For example, a process P_* waiting for acknowledgments for its "customized" request may receive another "customized" request from a process in the different group, instead of an acknowledgment to its own request. This scenario can happen when P_*'s sent message becomes content-broken, and thus P_* cannot receive acknowledgements from all processes in the different group. Because of subtle scenarios in Protocol 3 such as the one described above, the implementer of the protocol will gain the benefit of automatic analysis of the implemented protocol using TO-abstraction. We use one possible implementation of the protocol as our first case study in the paper.

3. Settings and Template

In this section, we explain the settings of a distributed real-time system that we assume and the syntax template that describes a restricted subclass of the ordinary TIOA guarded-command-style language. The basic idea of the template is to restrict use of time data and timeouts to a special form for which TO-abstraction can be applied. There is no restriction imposed for "untimed" part of the language.

3.1. Processes and Modules

A system consists of a set of processes $\{P_i\}$ and channels between processes $\{C_{i,j}\}$.[4] Processes in the system can be heterogeneous – a process can execute a different program from that of another. For example, processes may be split into two groups, Severs and Clients: Processes in Servers run the same program, and processes in Clients run a program that is different from the server, but is the same for all Clients. Processes communicate with each other via channels. A channel has a first-in first-out (FIFO) buffer in it to store process messages. The buffer size is bounded (in order to conduct model-checking after the abstraction), and a message sent to the channel when its buffer is full is simply discarded. We assume that processes are *loosely synchronized*: for any pair (P_i, P_j) of processes, the deviation between the values of

[4]In the full version of this paper [13], We actually give the user an option to use additional mudules, environment and helper modules, to describe a more general setting. Due to space limitation, we only talk about processes and channels in this paper.

their local clocks, $clock_i$ and $clock_j$, respectively, are bounded by an a priori known amount ε, as shown in the following inequality: $|clock_i - clock_j| < \varepsilon$. We assume that the ε bound is known to every process as its parameter. We also assume that a constant positive real value u is known to every process as its parameter. A TIOA that models a process in the system must have the following automation declaration: "Automaton P_i(ε, u: NonNegReal, p: OtherDiscreteParameters)". In the above declaration, p is a parameter that contains non-timing-ralated information, such as value initializations for "untimed variables". We discuss untimed and timed variables in Section 3.3.

The entire system is represented by composing processes and channels in the sense of TIOA (an automata composition by synchronizing output and input actions with the same name. See [7] for more details). COMPOSITION below represents the composition.

$$S(\varepsilon, u, \{init_i\}) = (\Pi_{i \in Process\_IDs} P_i(\varepsilon, u, init_i)) \times (\Pi_{i,j \in Process\_IDs} C_{i,j}) \quad \text{(COMPOSITION)}$$

The set $\{init_i\}$ in $S(\varepsilon, u, \{init_i\})$ represents a list of initialization parameters for each process P_i. Note that every process shares the same ε and u after the composition (ε and u are parameters of the entire system S).

3.2. *Action Signatures of Process P_i*

In this subsection, we present the form for the signatures (interfaces) of actions for process P_i. The system has output and input actions for message communication between processes, such as 'send' and 'receive'. A 'send' action has the following signature: send_i(j:ProcessID, M:UntimedMessage, χ:NonNegReal, κ:InteractionInst). This action represents that P_i sends real-valued "time data" χ and "untimed data" M to process P_j. We will explain in Section 3.5 the form of time data allowed in the template. Untimed data is an arbitrary value in a bounded domain. An interaction instance κ is a special identifier of process interactions to distinguish different set of interactions. A 'receive' action has the same type of interface as a 'send', in order to match communications between processes. A 'broadcast' action, bcast_i, has a similar interface as send_i, but instead of a process ID j, it has a set of process IDs J that represents a subset of processes to which the broadcast is performed. A process also has internal actions which are performed without communicating outside of the world.

A timeout action of P_i is modeled as an output action, $timeout_i^k$, and has a very specific form for its precondition (the transition guard) so that the timeout happens at the time the local clock of P_i hits a specified timeout time. The actual form of the precondition is described in Section 3.5.

To give the reader information about how templates are actually used in a real example, we present the Tempo code for our implementation of 3 that we use for one of the case studies. Due to space limitation, we present in Figure 1 only parts of the code. We explain some use of time data and timeout in this code. bcast$_i$ of 'CONST' in Figure 1 uses $clock_i + u$ for its broadcasting time data. It's timeout time is set to the same value. Whereas, bcast$_i$ of 'CUSTOM' picks a new time nonce, and broadcasts it. It's timed variable proposing_time_nonce is set to this time nonce, but the timeout is not set yet (since it has not obtained acknowledgments. The timeout time is updated using the time-stamp-estimation trick and a 'max' operation when a process receives a "BROKEN" message.

```
*   timeout_time is the only timeout variable $t_i^k$ in this process.
*   proposing_time_nonce is the only timed variable $x_i^k$ in this process,
      and stores a time nonce used for its own "customized" request.

output bcast$_i$(opponent_group,
              'CONST', $ts\_u$, $\perp$)
    pre   $ts\_u$ = clock$_i$ + $u$ $\wedge$
          pc = crit_ready
    eff   pc := crit;
          timeout_time := $ts\_u$;
          timeout_is_set := true;
output bcast$_i$(opponent_group,
              'CUSTOM', $tn$, $\kappa$)
    pre   $tn$ = new_time_nonce() $\wedge$
          pc = crit_ready $\wedge$
          $\kappa$ = new_interact_inst()
    eff   pc := waiting_for_ack
          proposing_time_nonce := $tn$
          my_interact_inst := $\kappa$
input receive$_i$($j$, 'BROKEN', 0, $\perp$)
    eff   if (pc = idle $\vee$
              pc = waiting_for_timeout $\vee$
              pc = sending_ack) then
              ...
            timeout_time :=
              max(timeout_time,
                  clock$_i$ + u + 2$\varepsilon$);
            timeout_is_set := true;
          fi;

input receive$_i$($j$, 'CUSTOM', $tn$, $\kappa$)
    eff   if pc $\neq$ crit then
            waiting_for_msg[j] := false
            %% resetting
            proposing_time_nonce := 0;
            ack_rcvd_list := $\emptyset$;
            if $tn + \varepsilon >$ clock$_i$ then
              pc := sending_ack;
              ack_interact_inst[j] := $\kappa$;
              timeout_time :=
                max(timeout_time, $tn$ + $\varepsilon$);
              timeout_is_set := true;
              ack_sending_list :=
                   ack_sending_list $\cup$ $j$;
            fi fi

output timeout$_i$
    pre   clock$_i$ = timeout_time $\wedge$
          timeout_is_set
    eff   timeout_time := 0;
          timeout_is_set := false;
          if pc = crit then
            pc := idle;
            ....
          fi
```

Fig. 1. Parts of Tempo code for the resource sharing protocol, Protocol 3.

3.3. *State Variables of Process P_i*

State variables in a TIOA that represents process P_i are split into the six groups shown in Table I. This explicit split of variables is the core of why we can apply TO-abstraction to the template.

Group Name	Variables	Types of Variables
Timed variables	$\{x_i^k\}_{k=1}^{\ell_i}$	NonNegReal
Timeout variables	$\{t_i^k\}_{k=1}^{r_i}$	NonNegReal
Timeout setting Booleans	$\{timeout\_is\_set_i^k\}_{k=1}^{r_i}$	Boolean
Interaction Instance variables	$\{w_i^k\}_{k=1}^{q_i}$	Integer
Untimed variables	$\{v_i^k\}_{k=1}^{m_i}$	BoundedDomain$_i^k$
Local clock variable	$clock_i$	NonNegReal

TABLE I
STATE VARIABLES OF PROCESS P_i

The first group consists of timed variables. These variables can store real-valued time data such as time nonces and time stamps (plus u), or the result of a 'max' operation of multiple pieces of time data. We will see the form of value assignments that can be used for timed variable in Section 3.5. The second group consists of timeout variables. These variables are special variables to set a time when a timeout action must be performed. They can store time data, possibly plus the slack ε, and the result of the time-stamp-estimation trick. The third group consists of timeout setting Booleans. These are Boolean variables that are used to indicate whether or not the timeout for $timeout_i^k$ is set. The fourth group consists of untimed variables. These variables can store any bounded-size information other than time data. An untimed variable can be, for example, a Boolean, a (bounded-size) counter, or a finite set of process IDs to aggregate the information about which processes have responded to P_i's message. The last group consists of just one variable, $clock_i$, which represents the local clock of process P_i. The value of this variable evolves over real time. The actual evolution is defined by the *trajectory* definition discussed in Section 3.4.

3.4. Trajectory of Process P_i

We insist that a TIOA model for process P_i have the following trajectory definition for evolution of the value of its local clock. The definition describes that 1. A local clock is monotonically increasing, and 2. When a local clock hits one of t_i^k (a timeout time) and the timeout for t_i^k is set, the timeout action must be performed before time elapses.

evolve: $d(clock_i) > 0$
stop when: $\exists k : clock_i \geq t_i^k \wedge timeout\_is\_set_i^k$

Note that the loose-synchronization assumption described earlier in this section cannot be described by the trajectory definition of one process, but is used in the soundness theorem as an assumption.

3.5. Transitions of Process P_i

In this subsection, we describe the template for transitions of process P_i. A transition is defined using its precondition (the guard) and effects. We have templates for both preconditions and effects. The template is defined formally using a BNF-like form. Due to space limitation, we cannot list the entire template in the paper. Instead, we explain important constructs in the template in this subsection. (See [13] for the entire template.)

Timed Expression Template: The templates for $TimeDataExpression$, $TimeoutExpression$, and $TimeoutUpdateExpression$ shown in TimedExpression are the most important templates in the entire template set. These includes a time nonce picking (new_time_nonce(), which computes a value greater than $clock_i$, a future time), computation of a time stamp plus the constant u ($clock_i + u$), a 'max' operation, timeout setting using ε, and the time-stamp-estimation trick ($clock_i + u + 2\varepsilon$). These expressions are used in templates for process transitions explained later in this section. See Figure 1 and its explanation again to observe that timed expressions in the code can be expressed using these templates.

$$\begin{aligned}
TimeDataExpression ::= {} & new\_time\_nonce() \mid clock_i + u \mid \\
& x_i^k \mid \chi \mid \\
& \max(TimeDataExpression, \\
& \quad TimeDataExpression) \\
TimeoutExpression ::= {} & TimeDataExpression \mid \\
& TimeDataExpression + \varepsilon \mid \\
& clock_i + u + 2\varepsilon \mid \\
& \max(clock_i + u + 2\varepsilon, \\
& \quad TimeDataExpression + \varepsilon) \\
TimeoutUpdateExpression ::= {} & TimeoutExpression \mid \\
& \max(t_i^k, TimeoutExpression)
\end{aligned} \qquad \text{(TimedExpression)}$$

Precondition of Timeout Actions: The precondition of timeout action $timeout_i^k$ must always have the following form: $clock_i = t_i^k \wedge timeout\_is\_set_i^k$. The above condition states that a timeout action $timeout_i^k$ is enabled when the local clock of P_i hits the specified timeout time t_i^k, and also the value of $timeout\_is\_set_i^k$ is true.

Precondition of 'send$_i$' and 'bcast$_i$' Actions: Preconditions for $\mathrm{send}_i(j, M, \chi, \kappa)$ and $\mathrm{bcast}_i(J, M, \chi, \kappa)$ must include the binding of the sending time data χ in the following form: $\chi = TimeDataExpression$. Basically, a process can include any time data (which can be expressed by the template) in its sending messages.

Effects of Actions: The effects of an action are constructed by simple assignments to variables and if statements that include assignments or nested if statements.

The user can use *expiration checks* in the branch condition of if statements. These checks are in the form of $X \sim clock_i$ or $X + \varepsilon \sim clock_i$, where $\sim$ is either $>$ or $\leq$, and X is either x_i^k or χ (χ is only for send, broadcast, and receive actions). Using an expiration check, a process can, for example, check whether received time data represents a future time, and thus can be validly set the time data as a timeout time (receive$_i$ of 'CUSTOM' in Figure 1 uses this check).

The values of untimed variables are assigned by arbitrary expressions that only use untimed variables. The values of timed variables are assigned by any form of $TimeDataExpression$, defined in TimedExpression.

A timeout is set and reset by $TimeoutAssignment$ defined in TIMEOUTASSIGN. $TimeoutUpdateExpression$ is defined in TimedExpression. We can see in TIMEOUTASSIGN that an assignment to a timeout variable t_i^k is always performed with a setting to $timeout\_is\_set_i^k$.

$$\begin{aligned}
TimeoutAssignment ::= {} & t_i^k := TimeoutUpdateExpression; \\
& timeout\_is\_set_i^k := true; \mid \\
& t_i^k := 0;\ timeout\_is\_set_i^k := false;
\end{aligned} \qquad \text{(TIMEOUTASSIGN)}$$

4. TIMEOUT ORDER ABSTRACTION

Timeout order abstraction (TO-abstraction) enables the user to abstract a TIOA model described using the template explained in Section 3 into an untimed finite-state model. This abstraction is performed as a code-to-code conversion from a model described by the template into ordinary (untimed) I/O automata code. We first explain the overview of this technique.

4.1. Intuition behind Timeout Order Abstraction

Our goal for TO-abstraction is to obtain an untimed model that conservatively abstracts the original one. In order to obtain an untimed model, we remove the local clocks of processes, and maintain just the right amount of information that can retrieve correct timeout orders in the original timed model. The key idea is to *not* focus on real values of time nonces and time stamps, but instead use identifiers (IDs) of them (labels in other words). Identifiers are integers that distinguish different time nonces and stamps. We explain in more detail how IDs can be used using toy examples described in Section 2.

In Protocol 1, time nonces are used to synchronize processes behavior. Processes control the ordering of timeouts using the relative difference of their timeout times measured on their local clocks: one process P_i sets a timeout to a time nonce TN_k, and the other P_j sets it to $TN_k + \varepsilon$. Information that is sufficient to retrieve the fact that P_j times out after P_i has done so in any possible scenario, is that both processes use the same time nonce TN_k, and P_i's timeout is set by adding the "slack" ε to the time nonce, but P_i's timeout is set solely by the time nonce. To keep track of the above information, we can use the following abstraction: Picks of real-valued time nonces are replaced by picks of symbolic IDs of them. The k-th picked time nonce TN_k in an execution of the original system is represented by ID (integer) k. The timeout time for P_1 set to TN_k is symbolically represented by a pair $(k, false)$, and the timeout time for P_2 set to $TN_k + \varepsilon$ is symbolically represented by a pair $(k, true)$ – the Boolean values represent whether the slack ε is added or not. Then, we remove the local clocks of both processes. After this abstraction, even though we cannot access to the information of local clock values (since they are removed), we can carefully choose which timeouts *must not occur* at the current state of the untimed model, by looking at the symbolic representations: If P_i's timeout is set to $(k, false)$ and P_j's timeout is set to $(k, true)$, P_j's timeout must not be performed, considering timeout orders that can possibly appear in the original timed model. The above discussion is the basic idea of time data abstraction and timeout order constraining, two of the three techniques that we use in combination for TO-abstraction.

We can apply the above idea of maintaining information that is sufficient to retrieve correct timeout orders to Protocol 2 as well. In this case, we use IDs of time stamps. Since processes always add u to time stamps, we can just keep track of the ID of a time stamp when the value computed from a time stamp plus u (in the form of $TS_k + u$) is communicated. We explain how we treat the time-stamp-estimation trick in the abstraction after we explain the abstraction for the max operation. In Protocol 3, processes use 'max' operations to conservatively update the timeout estimates. To represent a 'max', we use a set of IDs. For example, $\max(tn_1, tn_3, tn_7)$ is represented by $\{1, 3, 7\}$. Now tn_3 is considered as $\max(tn_3)$, and thus is represented by $\{3\}$. Timeout setting $\max(tn_1, tn_3) + \varepsilon$ is represented by $(\{1, 3\}, true)$. We can constrain timeout orders as follows: If for two sets of time nonce IDs IDs_1 and IDs_2, $IDs_1 \supseteq IDs_2$ and two timeouts are set at $(IDs_1, true)$ and $(IDs_2, false)$, respectively, then the timeout at $(IDs_1, true)$ is disabled. This is because IDs_1 represents a larger value than IDs_2. When time nonces and time stamps are used in the same protocol, we use two sets of integers (IDs) to represent one time data: $(\{1, 2\}, \{4\})$ represents $\max(TN_1, TN_2, TS_4 + u)$, and $(\{3\}, \{2\}, true)$ represents a timeout set at $\max(TN_3, TS_2 + u) + \varepsilon$. We do not allow a timeout update using a slack-added timeout and non-slack timeout, since the updated timeout cannot be represented symbolically.[5] We add a run-time check of consistency of timeout update in the abstraction, so that the abstracted model conservatively approximates the original.

The time-stamp-estimation trick using $clock_i + u + 2\varepsilon$ is abstracted as $(\{\}, picked\_ts\_id\_set, true)$, where $picked\_ts\_id\_set$ is the set of time stamp IDs that have been picked thus far in the current execution. $(\{\}, picked\_ts\_id\_set, true)$ represents $\max(picked\_ts\_set) + u + \varepsilon$ in the original model, where $picked\_ts\_set$ is the set of time stamps picked thus far. This reflects the discussion for the time-stamp-estimation trick in Section 2 that $clock_i + u + 2\varepsilon$ can be interpreted as $(clock_i + \varepsilon) + u + \varepsilon$, and $clock_i + \varepsilon$ over-approximates every time stamp that has been picked in physical time.

4.2. Overview of Timeout Order Abstraction

We use three sub-techniques in combination for timeout oder abstraction of a TIOA model described by the template explained in Section 3. The three techniques are: 1. Time data abstraction, 2. Timeout order constraining, and 3. Time data reuse and compression. The first two techniques are used to abstract the underlying real-time system into an untimed, but infinite-state model. The third technique is used to represent the infinite state space of the untimed abstraction using a finite one.

[5]This restriction is not too restrictive because the user can use two timeout variables for a situation in which one process P_1 is waiting for a timeout of another process P_2 (P_1 uses a slack-added timeout in this case) and at the same time is waited by a different process P_3 (P_1 uses a non-slack timeout, and P_3 uses a slack-added timeout).

We first explain each technique briefly here, and will go into more details in subsections of this section.

[Time Data Abstraction]: First, *time data abstraction* abstracts away local clocks from a given TIOA model, and abstracts real-valued time data in the system using symbolic representations, as discussed in Section 4.1. In addition, all boolean conditions in the original model that require a local clock value to determine their truth-values are conservatively approximated.

[Timeout Order Constraining]: Using information from the symbolically represented timeout time, we constrain the order of timeouts, as we discussed in Section 4.1.

[Time Data Reuse and Compression]: An execution of the untimed model may require infinite number of new time nonce and/or time stamp IDs because the length of a model execution is typically infinite. Our basic strategy is to *reuse* a time stamp or a time nonce that is once taken by some process but is no longer used anywhere in the model.

4.3. Time Data Abstraction

By *time data abstraction* that we describe in this section, the user can abstract a TIOA model described using the template discussed in Section 3 into an untimed infinite-state model. We will consider how we can make this untimed model a finite-state model in Section 4.5.

The basic idea of time data abstraction is to represent time stamps and time nonces using positive integers that represent the "identifiers" of time stamps and time nonces, as discussed in Section 4.1. The type of timed variables (which store time data) after abstraction becomes: $\mathsf{SymbolicTimeData} = (\mathsf{set\ of\ Natural}) \times (\mathsf{set\ of\ Natural})$, where the first set represents time stamps and the second represents time nonces. And the type of timeout variables becomes: $\mathsf{SymbolicTimeout} = (\mathsf{set\ of\ Natural}) \times (\mathsf{set\ of\ Natural}) \times \mathsf{Boolean}$, where the last Boolean entry represents whether or not the "slack" value ε is added to the timeout value. For the above two types SymbolicTimeData and SymbolicTimeout, we define two functions ts_set and tn_set that get the first integer set and the second integer set, respectively, from a given symbolic representation of time data or a timeout value. We also use a predicate (a Boolean function) slack_added for SymbolicTimeout to access to the third entry (a Boolean value) of SymbolicTimeout.

To use symbolic representation of time data discussed above, time data abstraction conducts the following five processes.

1) Over-approximating the preconditions of timeout actions,
2) Symbolically representing of time stamps and time nonces using IDs,
3) Abstracting expiration checks, and
4) Abstracting the time-stamp-estimation trick, $clock_i + u + 2\varepsilon$, and
5) Removing the local clock and the trajectory definition.

The transformation is formally defined as a function (a code-to-code conversion) from the TIOA syntax template defined in Section 3 to the ordinary I/O automata syntax. Due to space limitation, we cannot present the definition of this function in this paper, but the definition is described in [13]. In the following, we explain important parts of the abstraction.

Over-approximating the Preconditions of Timeout Actions: The precondition of every timeout action, which has the form $clock_i = t_i^k \wedge timeout\_is\_set_i^k$, is over-approximated by replacing it with $timeout\_is\_set_i^k$.

Symbolically Representing Time Stamps and Time Nonces: We treat time nonce IDs using a global variable $picked\_tn\_id\_set$ of positive integer type. $picked\_tn\_id\_set$ represents the set of time nonce IDs picked thus far. Now new_time_nonce() function gives the smallest ID not in $picked\_tn\_id\_set$. We perform book-keeping for time stamp IDs similarly using $picked\_ts\_id\_set$. Every appearance of $clock_i + u$ is replaced by new_time_stamp_ID(), which gives the smallest ID not in $picked\_ts\_id\_set$.

Abstraction of Expiration Checks: Expiration checks of the form $clock_i > x + \varepsilon$ is abstracted using the same logic as we use to determine the order of timeouts, explained in Section 4.4. Informally, if there is a timeout set at y and the value of y is at most x, then $clock_i > x + \varepsilon$ will not hold, considering the loose-synchronization assumption. If we cannot infer the truth value of the expiration check, then the abstraction uses a non-deterministic choice for the result of the expiration check.

Expiration checks of the form $clock_i > x$ is approximated using a special list, called the expired time-data list, which stores time nonce and time stamp IDs picked thus far that are "strongly expired" – expired on *all* local clocks. Whether a time nonce/stamp ID ID_k is strongly expired is deduced from either of the following two facts: 1. A timeout is performed at $TD + \varepsilon$, and symbolic time data TD contains ID_k; 2. A non-deterministic choice for $clock_i > x + \varepsilon$ results in evaluating the inequality to be true, and x includes ID_k. Using the expired list, we can infer the truth value of $clock_i > x$ in the following case: if all IDs in x are strongly expired, then $clock_i > x$ must be true. If we cannot infer the truth value of the expiration check, then the abstraction uses a non-deterministic choice.

Rule for the time-stamp-estimation trick: As we discussed in Section 4.1, the time-stamp-estimation trick using $clock_i + u + 2\varepsilon$ is abstractly represented by
$(\{\}, picked\_ts\_id\_set, true)$.

Removing the local clock and the trajectory definition: After the untiming abstraction conducted above, the definition of the transitions of the untimed system does not use local clocks $clock_i$ at all. Thus, we can safely remove local clock variables and the trajectory definition.

4.4. Timeout Order Constraining

The untimed model after time data abstraction looses the control over timeout orders in the original timed model because the abstraction removes local clocks of processes. Therefore, we need to put the correct timeout orders back into the untimed model by constraining timeout orders using the information from symbolically represented timeout time. To constrain timeout order, we allow the abstract scheduler that resolves non-determinism in the untimed model to disable specific timeouts. The scheduler determines the next action to be performed from the set of all enabled actions in the current state. Whether a specific action is enabled or not is normally determined solely by its precondition. In addition to each precondition of actions, for timeout actions, the scheduler looks at symbolically represented timeout time to decide whether a particular timeout action is enabled or not. Considering the loose synchronization assumption, if a timeout action $timeout^i_k$ is set to a time larger than the time for another timeout $timeout^j_\ell$ by more than ε, then $timeout^i_k$ occurs after $timeout^j_\ell$ has done so. By rephrasing the above statement in symbolic representations, we obtain the following. Timeout action $timeout^i_k$ of a process i is disabled (even when $timeout\_is\_set^k_i$ is true) when the following condition holds: $\exists t^j_\ell \in Timeout\_variables_j$: $(j \neq i) \wedge (\mathsf{tn\_set}(t^j_\ell) \subseteq \mathsf{tn\_set}(t^i_k)) \wedge (\mathsf{ts\_set}(t^j_\ell) \subseteq \mathsf{ts\_set}(t^i_k)) \wedge \neg\mathsf{slack\_added}(t^j_\ell) \wedge \mathsf{slack\_added}(t^i_k)$.

4.5. Time Data Reuse and Compression

The length of an execution of an untimed model obtained from time data abstraction can be infinite. This implies that an execution may require infinite number of new time nonces or time stamps. The simplest strategy that we can follow to reduce the necessary number of time nonces and time stamps is to *reuse* a time stamp or a time nonce once taken by some process but no longer used in any time data in the model. This reuse would not change the behavior of the untimed model (in terms of its traces) since a time nonce or a time stamp is symbolically represented by its IDs, and for the untimed model, these IDs have no more information than just distinguishing an ID from another. In addition, we sometimes[6] need a more elaborate technique that we call *time data compression* in order to effectively express multiple time nonces (or stamps) by one symbolic time nonce (or stamp). Please see [13] for more details of time data compression. This reuse and compression techniques do not completely exclude the possibility that the resulting model still needs an infinite number of time nonces or time stamps. However, by incorporating the monitor for availability of time nonces and time stamps, we can use the following strategy: we first bound the number of time nonces and time stamps that may used in the model, and when model-checking it, we also monitor the availability of time nonces and time stamps. If we obtain a counterexample for the availability, we increase the size of time nonces and time stamps. This "bound & supersize" technique is particularly effective for bounded model-checking. Time data reuse and compression is formally defined as a code-to-code conversion function in [13].

5. Soundness of Timeout Order Abstraction

In this section, we briefly describe how we can prove the soundness of TO-abstraction that we have discussed in Section 4. (A more detailed proof appears in [13].) The soundness claim we guarantee is stated as Theorem 1.

Theorem 1: For any execution α of the original timed model with any instantiation of ε and u, there exists a corresponding execution β of the untimed model such that values of the untimed variables in the state after the k-th (discrete) transition appearing in α are the same as those in the state after the k-th transition appearing in β.

The key technique to use is a *simulation relation* from the original model to the untimed abstraction [7]. The basic idea of the simulation relation technique is to find, for each possible transition and trajectory of one automaton, a corresponding transition that have the "same effects". For example, the maximum operation of the timed original model has the "same effect" of the symbolic maximum operation in untimed model. We split this simulation relation proof into two parts: 1. Simulation relation from the original model (we call the model $Orig$) to the infinite-state untimed model obtained after time data abstraction and timeout order constraint (we call this model Inf) and 2. Simulation relation from Inf to the finite-state untimed model obtained after time data reuse and compression (we call this model Fin).

Proof Idea: *[Simulation relation from $Orig$ to Inf]*: Essentially, the two models executes the same execution, one using numerical time data, and the other using the symbolic representation of it. The simulation relation is defined so that a state of $Orig$ and a state of Inf is related when the values of all untimed variables in $Orig$ are exactly equal to those in Inf, and also symbolic representations of time data in Inf "correctly" represent time data

[6]For example, we need this technique for the DHCP-F protocol. However, we do not need it for resource-sharing example described in Section 2.

in $Orig$. The notable difference between $Orig$ and Inf raises from two points. The first point is the difference of when timeouts can be executed in $Orig$ and Inf. In Inf, the local clocks are abstracted away, and instead, the scheduler conservatively disables timeouts that would violate the timeout order inferred by symbolic timeout times. We need to prove (and have actually proved in [13]) that when a timeout action is enabled in $Orig$, the same timeout action is also enabled in Inf. This fact comes from the observation that when the scheduler disables a timeout in Inf, the corresponding timeout in $Orig$ has not yet reached. The second point is from the over-approximation of '$clock_i + u + 2\varepsilon$' using $\max(picked\_ts\_id\_set) + u + \varepsilon$ in Inf. We take into account this point in the simulation relation by relating a state s of $Orig$ and a state r of Inf when the symbolic value of a timeout variable in r (when converted into the numeric value using the information in s) is not exactly equal, but is *less than* or equal to the value of the corresponding timeout variable in s. We have succeeded to prove a simulation relation with this condition (which is weaker than exact matching of symbolic and numerical values).

[Simulation relation from Inf to Fin]: A proof for a simulation relation from Inf to Fin is relatively easy compared to that from $Orig$ to Inf. The key idea is to add a table of corresponding time stamps (and time nonces) and reused/compressed version of them in Fin, so that Fin now emulates Inf in parallel to its usual execution. This table is just for proving correctness, and does not change the behavior of Fin. After this emulation is defined, we can easily relate Inf and Fin using the state values of Inf enumerated in Fin. □

6. Case Studies

In this section, we briefly present two case studies of TO-abstraction. The first one is an implementation of the resource-sharing protocol described as Protocol 3. The second one is the DHCP Failover protocol. We used the SAL model-checker [2] developed by SRI for both case studied. For the presented case studies, we manually abstracted the timed model into the untimed model described in the SAL language. We tried to define every data structure in SAL as general as we can, so that the structure can be reused for other case studies. The actual code can be found in [12]. We are planning to develop an automatic conversion tool for TO-abstraction. The full model-checking using BDD was not feasible for both case studies arguably due to the complexity of the protocol. (We encountered the out-of-memory error when SAL was computing the transition function, that is, even before verification.) Therefore, we conducted a bounded model-checking. The number of time stamps and time nonces were increased when the time-stamp/time-nonce unavailability error has occurred. All experiments are conducted using a Linux machine with an Intel Core^{TM} 2 Quad at 2.66 GHz and 4GB memory.

Resource-Sharing Protocol: We applied TO-abstraction to our implementation of Protocol 3 (partial code appears in Figure 1). We used the configuration of two Π_1 processes and two Π_2 processes. We verified the protocol up to depth 20, and found no counterexample. The verification time was 24915.5 seconds (6.9 hours).

DHCP Failover Protocol: We applied TO-abstraction to the DHCP-F protocol (we briefly explained the protocol in Section 2.) The main safety property of DHCP-F that we verify in this paper is the no-duplicated-address-assignment property – one specific address is assigned to at most one process. We used the minimum interesting configuration, which consists of one main server, one backup server, and two clients (for a possible duplicated assignment). We succeeded in verifying the protocol up to depth 20. The verification time was 350743.7 seconds (97.4 hours). To examine the effectiveness of bug-finding using TO-abstraction, we experimented with verification of DHCP-F with a slight change: We removed one condition for checking a sequence number to ensure that the received message is for the current round. After this mutation, we found a counterexample at depth 17 (the run time was 52851.28 seconds $\sim$ 14.7 hours). This counterexample has a complex scenario, and we believe that it is very hard to find by human analysis. More details of this counterexample appear in [13].

7. Conclusion

We presented *timeout order abstraction* (TO-abstraction), a technique to abstract a particular subset of loosely synchronized real-time distributed systems (LSRTDS's) into a finite-state untimed model. By using this technique, the user can model-check the untimed abstraction to verify the original system, or find bugs in it. TO-abstraction is (as far as we know) the first automatic analysis tool for time-parametric verification of LSRTD's. The user can use only maximum operation for manipulating time data in the current version of TO-abstraction. Extension of allowable operations is future study.

Acknowledgement: We thank anonymous reviewers for their helpful comments on a preliminary version of the presented paper.

References

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

[2] 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.

[3] S. Dolev, S. Gilbert, L. Lahiani, N. A. Lynch, and T. Nolte. Timed virtual stationary automata for mobile networks. In *Principles of Distributed Systems, 9th International Conference, OPODIS 2005*, volume 3974 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2006.

[4] R. Fan, I. Chakraborty, and N. Lynch. Clock synchronization for wireless networks. In *OPODIS 2004: 8th International Conference on Principles of Distributed Systems*, volume 3544 of *Lecture Notes in Computer Science*, pages 400–414. Springer, 2005.

[5] R. Fan, R. Droms, N. Griffeth, and N. Lynch. The DHCP failover protocol: A formal perspective. In *27th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems (FORTE 2007)*, volume 4731 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007.

[6] 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.

[7] 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.

[8] T. Nolte and N. Lynch. Self-stabilization and virtual node layer emulations. In *Stabilization, Safety, and Security of Distributed Systems, 9th International Symposium (SSS 2007)*, volume 4838 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2007.

[9] T. Nolte and N. Lynch. A virtual node-based tracking algorithm for mobile networks. In *International Conference on Distributed Computing Systems (ICDCS 2007)*, page 1. IEEE Computer Society, 2007.

[10] S. Umeno. Event order abstraction for parametric real-time system verification. In *EMSOFT 2008: The 8th ACM & IEEE International Conference on Embedded Software*, pages 1–10, October 2008.

[11] S. Umeno and N. Lynch. Supplimental files for the RTSS 2009 paper. The files can be obtained from `http://people.csail.mit.edu/umeno/to_abstraction`.

[12] S. Umeno and N. Lynch. Supplimental files for the submitted paper. The files can be obtained from `http://people.csail.mit.edu/umeno/to_abst`.

[13] S. Umeno and N. Lynch. Timeout order abstraction for formal verification of loosely synchronized real-time distributed systems. Technical report, Massachusetts Institute of Technology. To appear soon.

[14] S. Umeno and N. Lynch. Safety verification of an aircraft landing protocol: A refinement approach. In *Proc. of HSCC'07, Hybrid Systems: Computation and Control*, volume 4416 of *Lecture Notes in Computer Science*, pages 557 – 572. Springer, 2007.