Timeout Order Abstraction for Formal Verification of Loosely Synchronized Real-Time Distributed Systems (Draft)*

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

January 22, 2010

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

*This is a draft version of a technical report. Please do not distribute it.

[1] A supplemental information about this paper (the SAL code used for the case study and counterexample traces) can be obtained from the URL cited as [14].

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, [10, 11, 3, 4, 5, 15]). 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 the following interesting building blocks for processes to communicate time data in the protocol.

1. [Time nonce communication and timeout setting]: 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, under the loose synchronization assumption described earlier.

2. [Time-stamp-estimation trick]: As a special way of picking a time nonce, a process may use a time stamp (copied from the current clock value), plus a special constant waiting time. Timeouts using this special form can be used for a fault-tolerant function, what we call the *time-stamp-estimation* trick.

3. [Conservatively updating timeout estimates of other processes using a maximum operation]: A process may update its timeout time or variables by 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.

4. [Expiration check]: 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). The process can perform different transitions depending on the result of checks ('if' branches can use expiration checks).

Using the building blocks we described above, the user can describe protocols in which processes loosely synchronize their behavior using communications of timing-related information. We will see examples of what type of protocols can be modeled using the template in Section 2.

On the other hand, the user cannot describe protocols in the following categories.

1. There is a known upper bound on the message delay, and the system works correctly using this information about message delay bound. The IEEE 1394 root contention sub-protocol [12] and a timing-based at-most-once message delivery protocol described in [8] are of this type.

2. There is a known upper bound on process (or task) execution time for certain steps in the system, and the system works correctly using this information about process execution time. The biphase mark protocol [16] and the Fischer mutual exclusion algorithm ([9], Section 24.2) are of this type.

3. Processes can set timeouts at a time computed by complex algorithms or expressions (not just a 'max' operation), and the value of the computed time is important for correctness.

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. TO-abstraction uses an interesting conservative approximation for the special usage, which we will explain as the time-stamp-estimation trick. 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" a part 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.

Second, the template presented in the paper can present interesting building blocks for real-time protocols, and we demonstrate their use using examples. One way of viewing the template that presents restrictions to the more general Tempo language is that it restricts the uses of timing-related information to very particular ones. Another way of viewing the template is that it provides the protocol designers that do not have an experience of using timing-related information in their protocol design, with simple tools that can add interesting event-order guarantees using communications of time data. 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.

[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].

Third, we provide a case study on automatic exhaustive 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. We consider the complexity of formally verifying this protocol is considerably high because of the fact that a complete proof is not included in [5] because a proof consists of proving several non-trivial lemmas needed to prove the final correctness theorem. The authors state that they have proved each lemma using induction on the system execution length. Our case study provides exhaustive exploration of the 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. Because this is a bounded guarantee (with respect to the execution depth and the number of processes), it cannot replace a complete formal proof. However, considering the complexity of lemmas and theorem and their proofs in [5], one cannot be absolutely sure about the correctness of the proofs (unless one conducts a mechanical theorem-proving). In this regard, we believe that our case study helps the community gain more confidence for the correctness of DHCP-F.

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 [13]. 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 [13]. 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 their building blocks for time data communication 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 a process 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 prove the soundness of TO-abstraction. In Section 6.2, we illustrate the use of TO-abstraction by applying it to 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. DHCP-F 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. DHCP-F uses the loose-synchronization assumption to intelligently set processes' timeouts to perform desirable orderings of process timeouts. The protocol has interesting mutual-exclusion and fault-tolerance properties guaranteed by correct orderings of timeouts. This is the main reason we chose DHCP-F for this work, in order to extend our work for untiming abstraction of real-time system by focusing on key event orders [13].

We have found key building blocks of DHCP-F that other protocols can use under the loose-synchronization assumption. The loose-synchronization assumption that we use is that the skew between any two local clocks of processes in the system is strictly less than ε. 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. A typical way of using a time nonce to control a timeout ordering is as follows: A process P_1 picks a time nonce τ and sends it to another process P_2. By P_1 setting its timeout to τ, and P_2 setting its timeout to $\tau + \varepsilon$, P_2 times out after P_1 does so, considering the loose synchronization assumption. 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 for time nonces: a time nonce is valid if its value is larger than the current clock of the process that picks the time nonce.

2. The second way uses a *time stamp*, plus a special constant waiting time u. A time stamp is a value copied from the current value of the local clock of a process. Information about the duration of the above mentioned constant waiting time (the value of u) is shared by all processes. Timeout setting using a time stamp and u can be considered a special form of setting timeout using a time nonce (since a time nonce can be an arbitrary value that a process picks). This special form is used by a process to control the order of timeouts *without knowing when other processes time out*. This special timeout can be used when the system needs a fault-tolerance property. We will explain how this timeout order control can be used for fault-tolerance later in this section.

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 to time-share 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 access 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.

Protocol 1. Since P_2 knows that P_1 accesses the resource first, P_1 can immediately use the resource. When P_1 finishes its job, it sends a "done" message to P_2. Upon a receipt of P_1's "done" message, P_2 starts using the resource. When P_2 finishes, it sends P_1 a "done" message. P_1 and P_2 repeat the same routine forever.

Protocol 1 is inefficient in some cases because one process cannot start the job until it receives the other process's "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 2 shows. Protocol 2 makes more efficient use of the resource if the clock skew bound ε is smaller than the message delay.

Protocol 2. 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, and sends it to P_1. 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 messages sent between processes can be broken. When a message becomes broken inside the channel, processes receives a message that does not contain any meaningful contents. Under the above assumption, we cannot use Protocol 2 – if TN_1 sent from P_1 to P_2 is 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 arbitrary time nonce resolves this situation. Suppose processes share the value of the constant waiting time u.

Protocol 3. 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 2. 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.

The timeout setting using $clock_2 + u + 2\varepsilon$ in Protocol 3 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 message (which was broken) 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 the physical time*, and a process can do 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 4.

Protocol 4. 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 send 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: $timeout\_time := \max(received\_TS\_U + \varepsilon, timeout\_time)$. When a process receives a broken-content message, it uses the time-stamp-estimation trick, by setting a timeout at $\max(clock + u + 2\varepsilon, timeout\_time)$. 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 resource-sharing time, instead of the constant time u. Protocol 5 extends Protocol 4 by adding the above stated option.

Protocol 5. This protocol is similar to Protocol 4, but differs in the following aspect. When sending a resource-sharing time to another group, a process may choose to propose an arbitrary time using a time nonce, instead of the constant time 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 5 is subtle despite its relatively simple informal description. For example, we cannot determine from the informal description what a process P_* waiting for acknowledgments for its time-nonce resource-share request from other processes does when it receives another time-nonce resource-share 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. Therefore, we need a formal model for this protocol to specify every detail of reactions to all possible messages in every protocol state. We often see this kind of subtlety not only in real-time protocols, but also in untimed protocols. For untimed protocols, once we have a formal model of them, we can use

a conventional model-checker to exhaustively check all possible subtle scenarios of the protocol. TO-abstraction that we present in the paper enables the user to conduct, by model-checking, the same kind of exhaustive explorations of scenarios for particular loosely synchronized protocols, including the examples that we described above. We will show formal Tempo code for this protocol in Section 3.

3 Settings and Template

In this section, we explain the settings of a distributed real-time system that we assume and the restricted syntax template for the TIOA programming language with which a process in the system must be described in order to use TOA.

3.1 Processes and Modules

We assume that the system consists of the following components:

1. Processes: $\{P_i\}$
2. Channels between processes: $\{C_{i,j}\}$
3. Environment: V
4. Helper Modules: $\{H_\ell\}$

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. This is done by synchronizing send and broadcast actions of processes with channel input actions with the same name. 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.

A process P_i is parameterized with non-negative real values ε and u that we explain in the following. For this parameterization, a TIOA that models a process in the system has the following automaton declaration:

Automaton $P_i(\varepsilon$, 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.

We assume that processes are *loosely synchronized*: for any pair (P_i, P_j) of processes, the deviation between the values of 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 the constant waiting time length u is known to every process as its parameter.

The environment module is used to model input from the environment. Environment input is used, for example, to indicate a timing when a process encounters a failure or when a process recovers from a transient failure and resumes its operation.

A helper module models a module that can give processes auxiliary information about the current system execution. For example, a failure detector can give information about which

process has encountered a failure and who must be the next leader if the failed process was the current leader. Helper modules can also be modeled as a part of processes, but to model only functions of these modules, it is useful in certain cases to model them separately from processes.[3] For instance, a model of a failure detector as a helper module can observe the transitions of processes, and when detecting that a process encounters a failure, it can directly notify the failure information to other processes, without using an explicitly modeled channel.[4]

We insist that the environment or a helper module should not use a timing-related information to determine its behavior (transitions), but use only the current execution trace of the system and values of "untimed" variables in processes. We see the distinction between timed variables and untimed variables in a process in Section 3.3.

The entire system is represented by composing the above described components in the sense of TIOA (an automata composition by synchronizing output and input actions with the same name. See [7] for more details). COMPOSITION-1 below represents the composition.

$$\begin{aligned} S(\varepsilon, u, \{init_i\}) =& (\Pi_{i \in Process\_IDs} P_i(\varepsilon, u, init_i)) \times \\ & (\Pi_{i,j \in Process\_IDs} C_{i,j}) \times \\ & V \times (\Pi_{\ell \in Helper\_IDs} H_\ell) \end{aligned} \qquad \text{(COMPOSITION-1)}$$

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 is parameters of the entire system S).

A program in each process is described in a restricted subclass of the ordinary TIOA guarded-command-style programming language. In the rest of this section, we present the template for this restricted language.

3.2 Action Signatures of Process P_i

In TIOA, as well as in the ordinary I/O automata framework, actions (labeled transitions) are split into three groups: output, input and internal. Output and input actions with the same name are synchronized by an automata composition.

In this subsection, we present a template for the signatures (interfaces) of process 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 can be used to send a message that contains time data, untimed data, and an interaction instance. Time data stored in and sent by a process is constructed by very specific expressions described in $TimeDataExpression$ defined in Expression-1. This is the heart of the reason we can symbolically represent time data in the model described by the template. Untimed data is data stored in an untimed variable or some constant value that has the same type as the value of an untimed variable. Interaction instances are identifiers (or labels) of interactions between processes. A process may pick a new distinctive

[3] In this aspect, helper modules behave much like the environment, and could have indeed been combined with the environment module.

[4] An actual implementation of a failure detector may use communication with processes and some timing-related information. For instance, it may request a process to send a "heart-beating" message at a certain rate. However, a functional model of a failure detector can be constructed in the way described here.

interaction instance to indicate a new interaction sequence, or use an existing interaction instance stored in its state variable to indicate that the sending message is a continuation of an existing interaction sequence.

A timeout action is modeled as an output action, and has a very specific form for its precondition (the transition guard) so that the timeout happens at the time the local clock hits a specified timeout time. The actual form of the precondition is described in Section 3.5.2.

An environmental event such as a process failure and recovery is modeled as an input action from the environment. Similarly, an auxiliary information from a helper module is given to a process as its input action.

[Output actions]

1. $\text{bcast}_i(J : \text{SetOfProcesses},\ M : \text{UntimedMessage},\ \chi : \text{NonNegReal},\ \kappa : \text{InteractInst})$.
   This action represents that P_i broadcasts time data χ and untimed data M to a subset of processes J. (We explain time data and untimed data in Section 3.3.)

2. $\text{send}_i(j : \text{ProcessIndex},\ M : \text{UntimedMessage},\ \chi : \text{NonNegReal},\ \kappa : \text{InteractInst})$.
   This action represents that P_i sends time data χ and untimed data M to process P_j.

3. timeout_i^k.
   This action represents a timeout for the k-th kind timeout action of P_i.

[Input actions]

1. $\text{receive}_i(j : \text{ProcessIndex},\ M : \text{UntimedMessage},\ \chi : \text{NonNegReal},\ \kappa : \text{InteractInst})$.
   This action represents that P_i receives time data χ and untimed data M from P_j

2. env_input^k.
   This action represents the k-th kind environment input action.
   Examples: 'fail' and 'recover'.

3. hlp_input_ℓ^k.
   This action represents the k-th kind input from helper module H_ℓ.
   Example: 'lead' input from a failure detector.

[Internal actions]

1. internal_i^k.
   This action represents the k-th kind (untimed) internal action of P_i.

To give the reader information about how templates are actually used in a real example, we here present the Tempo code for our implementation of Protocol 5. Figures 1 and 2 show Tempo code for the Protocol 5.

Example 1. The first part shown in Figure 1 describes a declaration of types used for the protocol, the automaton declaration, the action signatures (interfaces for transitions), and state variables of the automaton with their initial assignments.

Processes in group Π_1 are colored RED, and those in Π_2 are colored BLUE.

Two message types, CONST and CUSTOM, are used to distinguish a time-sharing request using the constant time u and a request using a time nonce (a customized time). If the message content becomes broken, a process receives a message with BROKEN

The slack value ε and the constant waiting time u for the time-stamp-estimation trick are declared as automaton parameters. An automaton also has a parameter my_color that represents its color (RED or BLUE), and a parameter opponent_group that represents the different group of processes (process IDs of processes in the BLUE group for a RED process, and IDs of processes in the RED group for a BLUE process).

Interaction between processes use *interaction instances* (identifiers to distinguish different process interaction sequences). These interaction instances are accessed as κ parameter of 'send', 'bcast', and 'receive' actions.

The second part shown in Figure 2 describes discrete transitions and the trajectory (continuous evolution) of the automaton. Processes in the RED group use a resource first. Therefore the program counter, pc, of every RED process is initialized to crit_ready (in Figure 1), indicating that the process is ready to send a CONST or CUSTOM message.

The subtlety of the informal description of Protocol 5 has been resolved in this formal code by deciding exact reactions to a received message for every state. For example, as we can see in the definition of $\mathsf{receive}_i(j, \text{'CONST'}, ts\_u, \kappa)$ action, if a process receives a CONST message, it will transition to the "waiting-for-opponent-group-finish" mode (a timeout for the received time data is set, and pc is set to waiting_timeout) *even when that process was waiting for an acknowledgment for its own time-sharing request.* □

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 1. This explicit split of variables is the core of why we can apply TO-abstraction to the template.

Group Name	Variables	Type 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 1: State variables of process P_i

The first group consists of timed variables. These variables can store real-valued timing-related data such as time stamps and time nonces, or the result of a particular computation using pieces of timing-related data, which we will discuss in Section 3.5. We call timing-related data stored in a timed variable *time data*.

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 a specially extended form of time data, as we will see in Section 3.5.

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 interaction instance variables. These variables are used to store existing interaction instances received from process communications. These variables are distinguished from other untimed variables because as we described in Section 4.6.1, we reuse interaction instances not currently in use in order to use the state space efficiently.

The fifth group consists of untimed variables. These variables can store any bounded-size information other than time data. A value assigned to an untimed variable must be computed using only untimed constants and untimed variables. We will see the form of possible value assignments that can be used for untimed variable in Section 3.5 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, $clock_i$, that 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.

---

evolve:

$d(clock_i) > 0$

stop when:

$\exists k : clock_i \geq t_i^k \wedge timeout\_is\_set_i^k$

We cannot talk about the skew bound of ε by looking at just one process. Therefore, we consider the automata composition of processes $\{P_i\}_{i=1}^n$. After composing processes $\{P_i\}_{i=1}^n$, we assume that in any reachable state of the composed system, the system satisfies the following constraint:

$$\text{For every pair (i,j) of processes} : |clock_i - clock_j| < \varepsilon$$

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 and effects. We have templates for both preconditions and effects. The template is formally defined using a BNF-like form.

The template for transitions specifies the following important functions.

1. How processes can set their timeouts.
2. How processes can pick time nonces, time stamps, and combine them using 'max' operations.
3. How processes can use expiration checks of time data.
4. How processes can use interaction instances.
5. How processes can use the time-stamp-estimation trick.

3.5.1 Timed Expressions

The templates for $TimeDataExpression$, $TimeoutUpdateExpression$, and $TimeoutExpression$ shown in Expression-1 are the most important templates in the entire template set. We explain these three in the following. (The definition in Expression-1 is the version for effects for $\text{bcast}_i(J, M, \chi, \kappa)$, $\text{send}_i(j, M, \chi, \kappa)$, and $\text{receive}_i(j, M, \chi, \kappa)$ actions. The term χ in $TimeDataExpression$ is removed for effects for other actions and predicates of all actions.)

$$
\begin{aligned}
TimeDataExpression ::= {} & new\_time\_nonce() \mid clock_i + u \mid x_i^k \mid \chi \mid 0 \mid \\
& \max(TimeDataExpression, TimeDataExpression) \\
TimeoutExpression ::= {} & TimeDataExpression \mid TimeDataExpression + \varepsilon \mid \\
& clock_i + u + 2\varepsilon \mid \\
& \max(clock_i + u + 2\varepsilon,\ TimeDataExpression + \varepsilon) \\
TimeoutUpdateExpression ::= {} & TimeoutExpression \mid \max(t_i^k, TimeoutExpression)
\end{aligned}
\quad \text{(Expression-1)}
$$

1. $TimeDataExpression$ defines how a process picks a time nonce and a time stamp, or combine these using a 'max' operation. Function $new\_time\_nonce()$ computes a new time nonce, which is some arbitrary future time (on its local clock). We assume that when $new\_time\_nonce()$ is used more than one times in one transition (in its precondition and effects), it returns the same value.[5] The expression $clock_i + u$ represents picking a time stamp, and add the constant waiting time u to it. These two kinds of instances, time nonces and time stamps (plus u) are the two basic constructs of time data. A process may store or look up time data by accessing a timed variable x_i^k, or by looking at the received or sending time data χ. Two pieces of time data can be combined using the 'max' operation.

2. $TimeoutExpression$ defines how a process can set a timeout using time data that it has stored or received. The first two derivations for $TimeoutExpression$ describes the two basic timeout settings: The first one ($TimeDataExpression$) uses a piece of time data directly as a timeout time. Whereas, the second one ($TimeDataExpression + \varepsilon$) uses a piece of time data *plus the slack* ε as a timeout time. By using these two kinds of settings and communication of time data, processes can loosely synchronize their behavior in order for one group of processes time out after another group have done so, as we have explained in Section 2 using toy examples. The third one ($clock_i + u + 2\varepsilon$) is used for the time-stamp-estimation trick, which is used for Protocol 3 shown in Section 2. The last one, $\max(clock_i + u + 2\varepsilon,\ TimeDataExpression + \varepsilon)$, can be used to obtain a conservative estimate of other processes' timeout using stored time data and the time-stamp-estimation trick, at the same time.

3. $TimeoutUpdataExpression$ defines how a process can update a timeout-time value using $TimeoutExpression$. A timeout time can be updated using solely a new $timeoutExpression$, or using a 'max' of an existing timeout time t_i^k and a new $timeoutExpression$.

The above described three expressions are used in precondition and effects of actions described in the rest of this section. Here we show some of the examples of timed expressions.

[5] The user can use internal actions to pick multiple time nonces between their send or broadcast actions.

Example 2. A process can assign a value to its timed variables using $TimeDataExpression$. For example, by the assignment $x_i^1 := new\_time\_nonce()$, process P_i sets its first timed variable x_i^1 to a new time nonce value. P_i can also sets its timed variable x_i^2 to a new time stamp, plus the fixed waiting time u, by $x_i^2 := clock_i + u$. P_i can combine two time data by using a 'max' operation. For instance, P_i can update its accumulated estimate of the largest received time data, stored in x_i^3 thus far, by $x_i^3 := \max(x_i^3, \chi)$, where χ represents the received time data. P_i can also update time data stored in x_i^4 using x_i^3 and a new time nonce by $x_i^4 := \max(x_i^3, new\_time\_nonce())$.

P_i can set a timeout using time data, the time-stamp-estimation trick, or the combination of both, and this is expressed by $TimeoutExpression$. For instance, P_i can set a timeout for t_i^5 using time data stored in x_i^1 by $t_i^5 := x_i^1$ or $t_i^5 := x_i^1 + \varepsilon$. P_i can use the time-stamp-estimation trick for t_i^6 by $t_i^6 := clock_i + u + 2\varepsilon$, or can use the combination $t_i^6 := \max(clock_i + u + \varepsilon, x_i^4 + \varepsilon)$.

Finally, P_i can also update a timeout that is already set, by $TimeoutUpdateExpression$. P_i can, for example, update its timeout t_i^7 by $t_i^7 := \max(t_i^6, x_i^3 + \varepsilon)$. □

3.5.2 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. Process P_i is supposed to set the boolean variable $timeout\_is\_set_i^k$ to true at the same transition by which it sets the timeout time t_i^k.

3.5.3 Precondition of 'send' and 'bcast' Actions

Precondition for $\text{send}_i(j, M, \chi, \kappa)$ and $\text{bcast}_i(J, M, \chi, \kappa)$ must be described by $SendPre$ defined in Pred-1.

$$\begin{aligned} SendPre ::= {} & ChooseTime \wedge ChooseInteractInst \wedge UntimedPred \wedge ExpirationCheck \mid \\ & ChooseTime \wedge ChooseInteractInst \wedge UntimedPred \\ ChooseTime ::= {} & \chi = TimeDataExpression \\ ChooseInteractInst ::= {} & \kappa = new\_interact\_inst() \mid \kappa = w_i^k \mid \kappa = \perp \end{aligned} \quad \text{(Pred-1)}$$

In Pred-1, $ChooseTime$ is used to bind time data to be included in the sending message. This time data can be any time data expressed by $TimeDataExpression$ that we define later in Expression-1.

In Pred-1, $ChooseInteractInst$ is used to bind the interaction instance to be used for the sending message. Function $new\_interact\_inst()$ computes a new interaction instance that is different from any existing interaction instance in the current execution. The interaction instance for the sending message can be copied from a stored interaction instance in an interaction instance variable w_i^j. A special symbol $\perp$ can be used as a default interaction instance, when a process does not need a distinctive interaction instance.

In Pred-1 (and in the rest of this section), $UntimedPred$ is defined as follows as a combination of interaction instance checks and other untimed predicates. In the following template, we

restrict the use of interaction instances in such a way that they are used only for equality checks. This implies that the size of an interaction instance does not have any meaning, but whether one interaction instance matches another is the only way processes use interaction instances.

$$
\begin{aligned}
\mathit{UntimedPred} ::= {} & \mathit{InteractInstCheck} \mid \mathit{UntimedPredNoIntrInst} \mid \\
& \mathit{UntimedPred} \wedge \mathit{UntimedPred} \mid \mathit{UntimedPred} \vee \mathit{UntimedPred} \\
\mathit{InteractInstCheck} ::= {} & \kappa\ \mathit{EqualOrNot}\ w_i^k \mid w_i^k\ \mathit{EqualOrNot}\ w_i^{k'} \\
\mathit{EqualOrNot} ::= {} & = \mid \neq
\end{aligned}
\qquad \text{(Pred-2)}
$$

In Pred-2 (and in the rest of this section), $\mathit{UntimedPredNoIntrInst}$ is an arbitrary state predicate whose truth value can be determined only by looking at untimed variables $\{v_i^k\}$ (which do not include interaction instance variables $\{w_i^k\}$) in the current state of the model. Part of $\mathit{UntimedPredNoIntrInst}$ is used to bind the sending target(s) (j or J) and the untimed contents of the sending message (M) of a 'send' or 'bcast' action.

As we can see in Pred-1, an instance of $\mathit{ExpirationCheck}$ can be used for the precondition of a 'send' or 'bcast' action. The form of $\mathit{ExpirationCheck}$ is defined later in ExpCheck-1.

3.5.4 Precondition of Internal Actions

The precondition of an internal action must be constructed from $\mathit{InternalPre}$ defined in Pred-2.

$$
\begin{aligned}
\mathit{InternalPre} ::= {} & \mathit{ExpirationCheck} \wedge \mathit{UntimedPred} \mid \\
& \mathit{UntimedPred}
\end{aligned}
\qquad \text{(Pred-2)}
$$

3.5.5 Effects of Actions

The effects of an action must be constructed using the expressions shown in Assign-1.

$$
\begin{aligned}
\mathit{Assignments} ::= {} & (\mathit{Assignment})^+ \\
\mathit{Assignment} ::= {} & \mathit{TimeDataAssignment} \mid \mathit{TimeoutAssignment} \mid \\
& \mathit{InteractInstAssignment} \mid \mathit{UntimedDataAssignment} \mid \mathit{IfStatement} \\
\mathit{IfStatement} ::= {} & \text{IF } \mathit{IfCondition} \text{ THEN } \mathit{Assignments} \text{ FI} \mid \\
& \text{IF } \mathit{IfCondition} \text{ THEN } \mathit{Assignments} \text{ ELSE } \mathit{Assignments} \text{ FI} \\
\mathit{IfCondition} ::= {} & \mathit{ExpirationCheck} \mid \mathit{ExpirationCheck} \wedge \mathit{UntimedPred} \mid \mathit{UntimedPred}
\end{aligned}
\qquad \text{(Assign-1)}
$$

A value assignment to a variable in the effects statement of an action has four types: Time-data assignment, timeout assignment, interaction-instance assignment, and untimed-data assignment. These three types of assignments assign a value to a timed, timeout, and untimed variables, respectively. Untimed-data assignments (represented by $\mathit{UntimedDataAssignments}$) are arbitrary assignments (including multiple assignments using 'for' statements) that use only untimed variables $\{v_i^k\}$ and untimed constants (values that has the same type as values in untimed variables).

The forms of time-data and timeout assignments are defined in Assign-2.

$$\begin{aligned} TimeDataAssignment ::= \;& x_i^k := TimeDataExpression; \\ TimeoutAssignment ::= \;& t_i^k := TimeoutUpdateExpression;\ timeout\_is\_set_i^k := true;\ | \\ & t_i^k := 0;\ timeout\_is\_set_i^k := false; \end{aligned} \quad \text{(Assign-2)}$$

We can see in Assign-2 that an assignment of the timeout time (an assignment to a timeout variable t_i^k) is always performed with a setting to $timeout\_is\_set_i^k$. By doing this, we make sure that every time a timeout is set, $timeout\_is\_set_i^k$ becomes true, and when a timeout is reset to the default value 0, $timeout\_is\_set_i^k$ becomes false, indicating that no timeout is set for t_i^k.

InteractInstAssignment is defined in Assgn-3. An interaction instance variable w_i^k must be assigned to a sending or received interaction instance κ, or an interaction instance stored in another interaction instance variable.

$$InteractInstAssignment ::= \; w_i^k := \kappa \mid w_i^k := w_i^{k'} \quad \text{(Assgn-3)}$$

Processes can use *expiration checks* (defined in ExpCheck-1) for transition preconditions and if branch conditions in transition effects. By an expiration check, a process examines whether a specified timing-related information (time data or time data plus ε) has "expired" or not on its local clock – a process determines that a value is expired on its local clock if the value is strictly less than the current value of the local clock. A process can use this check, for example, to examine whether time data χ received from another process is a valid future time to set it as a timeout, by checking whether $\chi > clock_i$ or not. A process can also check whether $\chi + \varepsilon > clock_i$, in the case that the process wants to set a conservative timeout estimate for the received time data χ, and thus wants to check whether the value $\chi + \varepsilon$ has expired yet. Expiration checks are the only control flows that the user can use to branch the behavior of a process in one discrete transition depending on the values of stored, received, and sending time data.[6] We do not allow, for example, a branch control by comparing the sizes of two pieces of time data.

$$\begin{aligned} ExpirationCheck ::= \;& TimeDataExpression\ Compare\ clock_i \mid \\ & TimeDataExpression + \varepsilon\ Compare\ clock_i \\ Compare ::= \;& > \mid \leq \end{aligned} \quad \text{(ExpCheck-1)}$$

[Constraint for setting timeouts]: A process P_i must assign a value larger than its local clock value $clock_i$ to a timeout variable t_i^k. This is because, otherwise the timeout is set to the time already passed. In such a case, in this particular process model described in TIOA, the time cannot evolve from the time the timeout is set, and therefore gets stuck at a "time-lock" situation, similar to a deadlock. A process model can satisfy this constraint by ensuring the setting value is a future time using an expiration check, or by using *now_time_nonce*() or $clock_i + u$ (or a 'max' that includes either of them) which is guaranteed to be larger than $clock_i$.

[6] Timeout setting controls the behavior of processes with respect to when processes time out, but does not control how processes behave when processes time out. This is why we state that expiration checks are the only control flows that use time data, and change behavior of processes in one discrete transition. Note also that effects of a timeout action may include expiration checks.

3.6 Channels between Processes

As we have explained in Section 3.1, processes communicate (send, broadcast, and receive) data with each other via channels. Each channel has the first-in first-out (FIFO) buffer, and the size of the buffer is bounded.

Actions of channels are synchronized with processes' send, broadcast, and receive actions by the automata composition, and the sending and receiving data are passed from a process to a channel (send) or to multiple channels (broadcast), or passed from a channel to a process (receive).

The channel between process P_i and P_j is defined as $C_{i,j}$, and we use explicit distinction between timed data, untimed data, and interaction instance that a channel carries.

We refer to the time data stored in the k-th entry of the FIFO buffer of the channel $C_{i,j}$ as $cx^k_{i\to j}$. Similarly, we refer to the untimed data stored in the k-th entry of the FIFO buffer of the channel $C_{i,j}$ as $cv^k_{i\to j}$, and refer to the interaction instance for the same entry as $cw^k_{i\to j}$.

```
Color = enumeration of 'RED', 'BLUE'
Message = enuemration of 'CUSTOM', 'CONST', 'ACK', 'BROKEN'
ProgramCounter = enumeration of 'idle', 'waiting_for_ack', 'sending_ack',
                    'waiting_for_timeout', 'crit_ready', 'crit'

Automaton Time_Sharing_Process_i(ε, u: NonNegReal, my_color: Color, opponent_group: Process_IDs)
    signature
        output send_i(j: Process_ID, msg: Message, time_data: NonNegReal, κ: InteractInst)
        output bcast_i(J: Process_IDs, msg: Message, time_data: NonNegReal, κ: InteractInst)
        input receive_i(j: Process_ID, msg: Message, time_data: NonNegReal, κ: InteractInst)
        output timeout_i
    states
        clock_j: NonNegReal
        timeout_time: NonNegReal := 0
        timeout_is_set: Boolean := false
        pc: ProgramCounter := if my_color = RED then crit_ready else idle fi
        my_interact_inst: InteractInst := ⊥
        ack_interact_inst: Array Process_ID of InteractInst := [for (i: Process_ID): ⊥];
        poposing_time_nonce: NonNegReal := 0
        ack_rcvd_list: Set of Process_IDs := ∅
        ack_sending_list: Set of Process_IDs := ∅
        waiting_for_msg: Array Process_ID of Boolean := [for (i: Process_ID): true];
```

Figure 1: Tempo code for the time sharing Protocol 5. Part 1 of 2

```
transitions

output bcast_i(opponent_group, 'CONST', ts_u, ⊥)
    pre   ts_u = clock_i + u ∧ pc = crit_ready
    eff   pc := crit;
          timeout_time := ts_u;
          timeout_is_set := true;

output bcast_i(opponent_group, 'CUSTOM', tn, κ)
    pre   tn = new_time_nonce() ∧
          pc = crit_ready ∧
          κ = new_interact_inst()
    eff   pc := waiting_for_ack
          proposing_time_nonce := tn
          my_interact_inst := κ

input receive_i(j, 'CONST', ts_u, κ)
    eff   if pc ≠ crit then
            waiting_for_msg[j] := false;
            proposing_time_nonce := 0;   %% resetting
            ack_rcvd_list := ∅;   %% resetting
            if ts_u + ε > clock_i then
              if pc ≠ sending_ack
                then pc := waiting_for_timeout   fi;
              timeout_time :=
                max(timeout_time, ts_u + ε);
              timeout_is_set := true;   fi fi

input receive_i(j, 'CUSTOM', tn, κ)
    eff   if pc ≠ crit then
            waiting_for_msg[j] := false
            proposing_time_nonce := 0;   %% resetting
            ack_rcvd_list := ∅;   %% resetting
            if tn + ε > clock_i then
              pc := sending_ack;
              ack_interact_inst[j] := κ;
              timeout_time :=
                max(timeout_time, tn + ε);
              timeout_is_set := true;
              ack_sending_list := ack_sending_list ∪ j;
            fi fi

input receive_i(j, 'BROKEN', 0, ⊥)
    eff   if (pc = idle ∨ pc = waiting_for_timeout ∨
              pc = sending_ack) then
            ack_rcvd_list := ∅;   %% resetting
            waiting_for_msg[j] := false;
            if pc = idle
              then pc := waiting_for_timeout   fi;
            timeout_time :=
              max(timeout_time, clock_i + u + 2ε);
            timeout_is_set := true;
          fi;

output send_i(j, 'ACK', 0, κ)
    pre   pc = sending_ack ∧ j ∈ ack_sending_list ∧
          κ = ack_interact_inst[j]
    eff   ack_sending_list := ack_sending_list \ j;

input receive_i(j, 'ACK', tn, κ)
    eff   if pc = waiting_for_ack ∧
             κ = my_interact_inst then
            ack_rcvd_list := ack_rcvd_list ∪{j};
            if ack_rcvd_list = opponent_group
              then pc := crit;
                   timeout_time :=
                     proposing_time_nonce;
                   timeout_is_set := true;
            fi fi

output timeout_i
    pre   clock_i = timeout_time ∧ timeout_is_set
    eff   timeout_time := 0;
          timeout_is_set := false;
          if pc = crit then
            pc := idle;
            for (j: opponent_group):
              waiting_for_msg[j] := true;
            ack_rcvd_list := ∅
            proposing_time_nonce := 0
          elseif (pc = waiting_for_timeout ∨
                  pc = sending_ack) ∧
            (∀(j:opponent_group):¬waiting_for_msg[j])
    then
            pc := crit_ready;
            ack_sending_list := ∅;   fi

trajectories
    trajdef trajectory
        evolve    d(clock_j)> 0
        stop when   clock_j ≥ timeout_time ∧ timeout_is_set
```

Figure 2: Tempo code for the time sharing Protocol 5. Part 2 of 2

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. 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 2, 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 $TN_k + \varepsilon$. The right amount of information needed 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: Picking real-valued time nonces are replaced by picking 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 just the right information to retrieve correct timeout orders to Protocol 3 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 the time-stamp-estimation trick is treated in the abstraction after we explain the abstraction for the max operation.

In Protocol 4, 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 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 $\max(TN_3, TS_2 + u) + \varepsilon$.

The time-stamp-estimation trick using $clock_i + u + 2\varepsilon$ is abstracted into $(\{\}, picked\_ts\_id\_set, true)$, 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 Protocol 3 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.

We can see that the template for time data and timeout assignments (Assign-2) is carefully constructed so that any time data and timeout time settings can be represented by the above described symbolic representation. For example, we do not have a mixture of a slack-added timeout time and a non-slack timeout time, such as $max(tn_1, tn2 + \varepsilon)$.[7]

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 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 "compress" the infinite state space of the untimed abstraction into a finite one.

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, such as time stamps, 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-value 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. To do so, we allow the abstract scheduler that resolves non-determinism in the untimed model to disable specific timeouts, depending on their symbolic timeout time representations.

[Time Data 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. One simple strategy that could resolve this situation 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. We

[7] 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).

actually need a more elaborate scheme than just reusing time stamp/nonce IDs. We will discuss about this scheme in Section 4.6.

4.3 Time Data Abstraction (Overview)

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 4.6.

The basic idea of time data abstraction is to represent time stamps and time nonces by using positive integers that represent the "identifiers" of time stamps and time nonces. For example, we consider that the identifier of a time stamp ts_i that is taken as the i-th new time stamp in an execution of the original TIOA is i, and in the untimed model, ts_i is symbolically represented by this integer i.

As we have discussed in Section 4.1, any time data can be expressed symbolically as a pair of a set of time stamps and a set of time nonces. Thus, by representing time stamps and time nonces using integers, time data can be expressed as a pair of two integer sets. Similarly, from the observation that we discussed in Section 4.1 for timeout variable, the value of a timeout variable can be expressed as a triple that contains two sets of integers and one Boolean value that represents whether ε is added to time data or not.

The type of timed variables (which store time data) after abstraction becomes: SymbolicTimeData = (set of Natural) $\times$ (set of Natural), where the first set represents time stamps and the second represents time nonces. The zero value (0) for time data is symbolically represented by $(\{\}, \{\})$, a pair of two empty sets of natural numbers. And the type of timeout variables becomes: SymbolicTimeout = (set of Natural) $\times$ (set of Natural) $\times$ Boolean (the last Boolean entry represents whether or not the "slack" value ε is added to the timeout value). The zero value (0) for a timeout value is symbolically represented by $(\{\}, \{\}, false)$.

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 use function time_data to obtain the time-data part (two sets) of SymbolicTimeout. The following function symb_max_part accesses the symbolic-time-data part in a timeout value.

$$\mathsf{symb\_max\_part}(t : \mathsf{Symbolic\_Timeout}) : \mathsf{Symbolic\_Time\_Data} := (\mathsf{ts\_set}(t), \mathsf{tn\_set}(t))$$

We use a predicate (a Boolean function) slack_added for SymbolicTimeout that represents whether the third entry (a Boolean value) of SymbolicTimeout is true or false. We use a predicate is_zero to check if the symbolic value of type SymbolicTimeData or SymbolicTimeout represents zero (namely, $(\{\}, \{\})$ for SymbolicTimeData and $(\{\}, \{\}, false)$ for SymbolicTimeout).

For two symbolic time data we define a partial-order $\preceq$ as follows:

$$td_1 \preceq td_2 \quad \text{iff} \quad \mathsf{tn\_set}(td_1) \subseteq \mathsf{tn\_set}(td_2) \wedge \mathsf{ts\_set}(td_1) \subseteq \mathsf{ts\_set}(td_2)$$

Now we explain how these symbolic representations of time data and a timeout value are used in the untimed abstraction.

We use the following symbolic maximum operation 'symb_max'.

$$\text{symb\_max}(symb\_td1, symb\_td2) = \\ (\text{ts\_set}(symb\_td1) \cup \text{ts\_set}(symb\_td2),\ \text{tn\_set}(symb\_td1) \cup \text{tn\_set}(symb\_td2))$$

We use the hat symbol '$\widehat{\ }$' to represent the symbolic version of timed and timeout variables and time data stored in channels. Namely, we have $\widehat{x_i^k}$ instead of x_i^k, $\widehat{t_i^k}$ instead of t_i^k, and $\widehat{cx_{i\to j}^k}$ instead of $cx_{i\to j}^k$. The type of the above variables are SymbolicTimeData, SymbolicTimeout, and SymbolicTimeData, respectively.

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 code-to-code-conversion function Υ from the TIOA syntax template defined in Section 3 to the ordinary I/O automata syntax, and is described in Section 4.4. In Sections 4.3.1 - 4.3.5 we explain informally how a transformation is conducted by time data abstraction.

4.3.1 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 $timeout\_is\_set_i^k$.

4.3.2 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*.

4.3.3 Abstracting Expiration Checks

The basic idea for abstracting expiration checks is to use non-deterministic choice to decide the results of the checks by default, but use a concrete truth value when the truth value of the check can be implied by the history of the current execution. To keep the history information needed to imply the truth value of certain expiration checks, we use a special list, the *expired time-data-ID list*. The expired time-data-ID list consists of the two sub-lists: the expired time-nonce-ID list and the expired time-stamp-ID list. These two sub-lists are maintained using state

variables expired_tn_ID_list and expired_ts_ID_list, respectively. Basically, when we can imply from an execution of the untimed model that the original numeric value τ represented by a time nonce or time stamp ID is *globally expired* (for all $clock_i$ in the system, $\tau < clock_i$), we add the ID to the expired time-data-ID list. The list is used to determine the truth values of expiration checks in the form of $TimeDataExpression_m \sim clock_i$ where $\sim$ is either $>$ or $\leq$. If we have $TimeDataExpression_m \preceq$ (expired_tn_ID_list, expired_ts_ID_list) in the state in which the above expiration check is performed, the truth value of $TimeDataExpression_m < clock_i$ must be true. This is because for every ID ID_* included in $TimeDataExpression_m$, the numeric value that ID_* represents is smaller than the current local clock value $clock_i$, and thus $TimeDataExpression_m$ that represents the maximum of these numeric values is also smaller than $clock_i$.

We cannot use the expired time-data-ID list to determine the truth value of expiration checks in the form of $TimeDataExpression_m + \varepsilon \sim clock_i$, since we cannot imply any definite fact about the above inequality just from the information about what IDs are globally expired. Therefore, if processes in the system does not use an expiration check of the form of $TimeDataExpression_m \sim clock_i$, we do not need the expired time-data-ID list in the untimed abstraction. Indeed, though the untimed abstraction for DHCP-F uses the expired time-data-ID list, the untimed abstraction of the resource-sharing example does not use the expired time-data-ID list since processes does not use an expiration check of the form of $TimeDataExpression_m \sim clock_i$.

We update the expired time-data-ID list in the following two ways: 1. When a timeout set at $\widehat{t_i^k}$ is performed, and $\widehat{t_i^k}$ uses a slack (slack_added($\widehat{t_i^k}$) is true), we add tn_set($\widehat{t_i^k}$) to expired_tn_ID_list and add ts_set($\widehat{t_i^k}$) to expired_ts_ID_list, and 2. When an expiration check of the form $TimeDataExpression_m + \varepsilon \sim clock_i$ is performed and non-deterministic choice for the result of the check infers that $TimeDataExpression_m + \varepsilon \leq clock_i$, for the symbolic time data $\widehat{\tau}$ that represents the value of $TimeDataExpression_m$, we add tn_set($\widehat{\tau}$) to expired_tn_ID_list and add ts_set($\widehat{\tau}$) to expired_ts_ID_list.

Suppose we have only the following two types of expiration checks in one transition: $TimeDataExpression_m >$ $clock_i$ and $TimeDataExpression_m \leq clock_i$, where $TimeDataExpression_m$ are the (syntactically) same time data expression. In such a case, we can use the following strategy to conservatively approximate expiration checks: We use an auxiliary Boolean variable time_data_already_expired to represent the result of the expiration check of the form of $TimeDataExpression_m \leq clock_i$. We look at the expired time-data list, and if $TimeDataExpression_m \preceq$ (expired_tn_ID_list, expired_ts_ID_list) at the current state, $TimeDataExpression_m < clock_i$ must be true, and thus we assign true to time_data_already_expired. Otherwise, we cannot say a definite fact about the truth-value of the expiration checks. Thus we use non-deterministic choice of true and false to assign the value of time_data_already_expired. After assigning time_data_already_expired, we use time_data_already_expired as the result of the check $TimeDataExpression_m \leq clock_i$. The result of $TimeDataExpression_m \leq clock_i$ is represented by the negation of time_data_already_expired. The above discussion gives us the following abstraction by code conversion:

Add:

---

time_data_already_expired := if $TimeDataExpression_m \preceq$ (expired_tn_ID_list, expired_ts_ID_list) then true else choose{true,false} fi;

To the very beginning of the transition effects.
And Replace:
$TimeDataExpression_m \leq clock_i$
With:
time_data_already_expired
And Replace:
$TimeDataExpression_m > clock_i$
With:
¬time_data_already_expired

The above conversio scheme works only for the case that we have only one kind of an expiration check, or two kinds of expiration checks where the two kinds are the negation of each other. If we have different types of expiration checks that use different time data expression and/or different slack-added conditions ($TimeDataExpression_m \sim clock_i$ or $TimeDataExpression_m + \varepsilon \sim clock_i$), we need a more elablate way of deciding results of expiration checks, as we describe in the following.

A process may use more than one expiration checks in one transition (possibly one in the precondition, and multiple checks in effects). In order to perform a non-deterministic choice that is consistent with respect to the same expiration checks appearing different places in the code for one transition, we use a vector of Boolean variables, *expiration-check vector*, in the following way: when a transition uses multiple expiration checks in its code, it first decide, by non-deterministic choice, the results of the entire expiration checks appearing in the code, and then use the results by referring to this vector.[8]

Example 3. Suppose one transition uses four expiration checks $\chi > clock_i$, $x_i^1 > clock_i$, $x_i^1 + \varepsilon > clock_i$, $x_i^2 \leq clock_i$ in multiple different places. In the untimed model, the process first decide the Boolean values in the expiration check vector v. Suppose v becomes $(true, false, false, true)$ after the non-deterministic choice. The results of the three kinds of expiration checks are referred as $v[1]$, $v[2]$, and $v[3]$, respectively. For instance, every appearance of $\chi > clock_i$ is replaced with $v[1]$, which is true in this case.

To index different forms of time data used for the expiration-check vector, we use the expiration-check-index information α. This information includes a mapping that maps a time data expression used in expiration checks to an index (integer label) of that expression. Given Tempo code described using the template, α is constructed as follows: All $TimeDataExpression$'s used in an expiration check in the form of $TimeDataExpression \sim clock_i$ ($\sim$ is either $<$ or $\leq$) are

[8] In this abstraction, we are considering the case that the value of the time data used for an expiration check performed in multiple places in the code does not change between different places. For example, we cannot use one boolean value to express the truth values of multiple expiration checks in the exact same form in the following case: one expiration check $x_i^k > clock_i$ is used in the very beginning of the transition, then the value of x_i^k is re-assigned, and $x_i^k > clock_i$ is performed at the end of the transition. If the value of time data used in multiple expiration checks may change, we use the default non-deterministic choice of true and false as the most conservative approximation.

labeled with integer indices, and the set of these indices are referred as $ExpirationCheckIndexWithoutSlack$. Function α_0 maps a given $TimeDataExpression$ to its index in $ExpirationCheckIndexWithoutSlack$. Similarly, all $TimeDataExpression$'s used in an expiration check in the form of $TimeDataExpression+ \varepsilon \sim clock_i$ ($\sim$ is either $<$ or $\leq$) are labeled with integer indices not in $ExpirationCheckIndexWithoutSlack$, and the set of these indices are referred as $ExpirationCheckIndexWithSlack$. Function α_ε maps a given $TimeDataExpression$ to its index in $ExpirationCheckIndexWithSlack$.

If one $TimeDataExpression$ is used in both of $TimeDataExpression \sim clock_i$ and $TimeDataExpression+ \varepsilon \sim clock_i$, that expression gets two indices, one for $TimeDataExpression \sim clock_i$, and one for $TimeDataExpression + \varepsilon \sim clock_i$.

Example 4. Code for one process uses $\chi > clock_i$, $x_i^1 > clock_i$, $x_i^1 \leq clock_i$, $x_i^1 + \varepsilon > clock_i$, and $\max(x_i^1, x_i^2) > clock_i$ for expiration checks.

An example of α_0 is: $\alpha_0(\chi) = 1$, $\alpha_1(x_i^1) = 2$, and $\alpha(\max(x_i^1, x_i^2)) = 3$. Note that the results of $x_i^1 > clock_i$ and $x_i^1 \leq clock_i$ are complementary, thus only one index is given to x_i^1.

An example of α_ε is: $\alpha_\varepsilon(x_i^1) = 4$. □

We also use $\overleftarrow{\alpha}$ that maps a given expiration check index to the original $TimeDataExpression$.

The information α consists of:

$$(\alpha_0, \alpha_\varepsilon, \overleftarrow{\alpha}, ExpirationCheckIndexWithSlack, ExpirationCheckIndexWithoutSlack).$$

The following function expiration_consistent checks whether the given Boolean vector represents a "reasonable" result set for the expiration checks, as we described earlier in this section. This version of expiration_consistent only checks whether the results of the expiration checks are consistent with what we can learn from the expired time nonce and time stamp ID lists and the current timeout settings: 1. If the expired ID lists includes all IDs in $TimeDataExpression_m$, then it implies that $TimeDataExpression_m$ has been globally expired (for all $clock_i$, $TimeDataExpression_m < clock_i$), and thus $v[m]$ must be true, if $v[m]$ represents the result of $TimeDataExpression_m < clock_i$ (if $v[m]$ represents the result of $clock_i > TimeDataExpression_m + \varepsilon$, we cannot imply any fact using information from the expired ID lists, and thus we do not constrain the value of $v[m]$); and 2. If there exists a timeout for some process set before $TimeDataExpression_m$, then $TimeDataExpresion_m + \varepsilon < clock_i$ must be false, considering the loose-synchronization assumption.

expiration_consistent(v : Array of Boolean) : Boolean =

$(\forall m \in ExpCheckIndexWithoutSlack : (\overleftarrow{\alpha}(m) \preceq (\text{expired\_tn\_list}, \text{expired\_ts\_list}) \Rightarrow v[m] = true)) \wedge$

$(\forall m \in ExpCheckIndexWithSlack :$

$\quad ((\exists \widehat{t_i^k} \in SymbolicTimeoutVar : \text{slack\_added}(\widehat{t_i^k}) \wedge \text{time\_data}(\widehat{t_i^k}) \preceq \overleftarrow{\alpha}(m)) \Rightarrow v[m] = false))$

The above function expiration_consistent does not take into account of the fact that the result of an expiration check in some cases implies another expiration check. For example, if a vector represents that $tn_3 > clock_i$ is true, then $\max(tn_3, tn_5) > clock_i$ must be true as well. We can express the constraints to v that take into account the above described fact using another version of expiration_consistent: In this version, for the first constraint described in the function,

instead of using only expired ID lists, we use the ID set that is the union of the expired ID lists and the IDs included in all $TimeDataExpression$'s for which $v[m]$ is true. This is because the value of each $v[m]$ represents whether the value is expired locally in that process or not, and therefore if $v[m] = true$, then all IDs included in $TimeDataExpression_m$ are considered expired in the underlying interpretation v of expiration checks. Moreover, we can use the same strategy to constrain $v[m]$'s that represent $TimeDataExpression_m + \varepsilon < clock_i$. In this case, instead of using the union of IDs in the expired lists and the IDs for $TimeDataExpression$ with $v[m] = true$, we can use IDs included in $TimeDataExpression$ such that $v[m] = true$ and $v[m]$ represents $TimeDataExpression_m + \varepsilon < clock_i$. By using this version of expiration_consistent we can avoid using an interpretation v that represents unrealizable results of expiration checks (for example $tn_3 > clock_i$ is true, but $\max(tn_3, tn_5) > clock_i$ is false).

Note that with respect to the soundness, it is valid to use any implementation of expiration_consistent as long as the constraint expressed by expiration_consistent includes all types of the results of expiration checks that could happen when time nonce or time stamp IDs are substituted by real values under the assumption of globally expired IDs given by the expired time data list. For example, the version of expiration_consistent that allows *any* combination of Boolean values for v is a sound implementation of expiration_consistent, since it includes all types of v, including unrealizable ones. Though using this version of expiration_consistent is sound, it may include a spurious counterexample from under-constraining the non-deterministic choice for expiration checks.

Using any version of a sound expiration_consistent implementation, we can use the following abstraction by code conversion:

Add:

---

expiration_check_vector := choose{v: ExpirationCheckVector | expiration_consistent(v)
}

To the very beginning of the transition effects.
And Replace:
$TimeDataExpression_m \leq clock_i$
With:
expiration_check_vector$[\alpha_0(TimeDataExpression_m)]$
And Replace:
$TimeDataExpression_m > clock_i$
With:
$\neg$expiration_check_vector$[\alpha_0(TimeDataExpression_m)]$
And Replace:
$TimeDataExpression_m + \varepsilon \leq clock_i$
With:
expiration_check_vector$[\alpha_\varepsilon(TimeDataExpression_m)]$
And Replace:
$TimeDataExpression_m + \varepsilon > clock_i$

With:

$\neg$expiration_check_vector$[\alpha_\varepsilon(TimeDataExpression_m)]$

The above conversion is formally defined as part of the definition of the time data abstraction function in Section 4.4.

4.3.4 Approximating 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
$(\{\}, picked\_ts\_id\_set, true)$.

4.3.5 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 Time Data Abstraction (Formal Definition)

In this subsection, we explain details of time data abstraction, the first technique used for timeout order abstraction. The abstraction is basically defined as an algorithm that converts code described using the template presented in Section 3 (a subset of the TIOA language) into the ordinary I/O automata language.

To conduct timeout data abstraction, we need to add five auxiliary variables picked_ts_ID_set, picked_tn_ID_set, expired_tn_ID_list, expired_ts_ID_list, and expiration_check_vector.

As we have briefly explained earlier in this section, the former four auxiliary variables are accessed (written and read) by multiple processes that use time nonces or time stamps. This means that the variables are global (or shared) variables. In the ordinary I/O automaton programming language, we cannot express a global or shared variable for processes that needs to be composed: the processes must communicate via synchronized actions. In this report, we use the extension of I/O automata with global variables, which comes from an idea similar to the one for the shared memory system modeling using I/O automata described in [9], Section 3.1.

The system has a set of global variables $G = \{g_1, g_2, \cdots, g_k\}$, and each variable g_ℓ has a value in a set $values_{g_\ell}$. A transition π of a process P_i is defined not as the set of ordinary triples of the form (s_i, π, s_i'), where s_i and s_i' are the local states of P_i, but as the set of triples of the form $((s_i, value\_vector), \pi, (s_i', value\_vector'))$, where $value\_vector$ and $value\_vector'$ are vectors of values of global variables in the set $\Pi_{\ell=1}^{k} values_{g_\ell}$. When π does not use any global variable, $value\_vector = value\_vector'$, and when π writes to some global variables, $value\_vector'$ expresses the changes from $value\_vector$.

We check the compatibility of automata to be composed by syntactically checking that no two automata accesses the same global variable (or the same entry in the case an array is shared), as well as checking the compatibility of interfaces of them, like done for ordinary I/O automaton.

A composition of these extended automata are performed in the vary same manner as the ordinary I/O automata, except that a state of the composed system is a vector of states of $\{P_i\}$ in $\Pi_i states(P\_i)$ plus the vector of values of global variables in $\Pi_{\ell=1}^{k} values_{g_\ell}$, and a transition changes global variables as follows: for internal actions or unsynchronized input or output actions, $((s_i, value\_vector), \pi, (s_i', value\_vector')$ for synchronized input and output actions,

Now we are ready to present a detailed description of time data abstraction. Time data abstraction is conducted by executing the following steps.

1. [**Automaton parameters**]

   $P_i(\epsilon, u : \text{NonNegReal}, p : \text{OtherDiscreteParameters})$
   becomes
   $Untimed\_P_i(p : \text{OtherDiscreteParameters})$

2. [**Data types for symbolic timed and timeout variables**]

   ADD:

   Vocabulary Untimed_Abstraction_vocab
   % Types for symbolic representation used in an untimed abstraction
   SymbolicTimeData = (set of Natural) × (set of Natural)
   SymbolicTimeout = (set of Natural) × (set of Natural) × Boolean
   % Number of the expiration check types in the process
   ExpirationCheckType = {n: NonNegInt | $n \leq$ expirationCheckSize}
   % Vector of Boolean used for expiration checks
   ExpirationCheckVector = Array ExpirationCheckType of Boolean

   ---

   where expirationCheckSize is the number of different expiration checks in the transition. This value is pre-computed, and is embedded in the above code.

   ADD:

   ---

   import Untimed_Abstraction_vocab

   ---

   in the "import" statement of the process.

3. [**Signatures**]

   $\text{bcast}_i(J : \text{SetOfProcesses}, M : \text{UntimedMessage}, \chi : \text{NonNegReal}, \kappa : \text{InteractInst})$
   becomes
   $\text{bcast}_i(J : \text{SetOfProcesses}, M : \text{UntimedMessage}, \widehat{\chi} : \text{SymbolicTimeData}, \kappa : \text{InteractInst}, v : \text{ExpirationCheckVector})$.

   $\text{send}_i(j : \text{ProcessIndex}, M : \text{UntimedMessage}, \chi : \text{NonNegReal}, \kappa : \text{InteractInst})$
   becomes
   $\text{send}_i(j : \text{ProcessIndex}, M : \text{UntimedMessage}, \widehat{\chi} : \text{SymbolicTimeData}, \kappa : \text{InteractInst}, v : \text{ExpirationCheckVector})$.

receive$_i$(j : ProcessIndex, M : UntimedMessage, χ : NonNegReal, κ : InteractInst)
becomes
receive$_i$(j : ProcessIndex, M : UntimedMessage, $\widehat{\chi}$: SymbolicTimeData, κ : InteractInst).

internal$_i^k$
becomes
internal$_i^k$(v : ExpirationCheckVector)

All other signatures remain the same.

4. [**States**]

   We use the hat symbol ' $\widehat{\ }$ ' to represent the symbolic version of timed and timeout variables.

   For timeout variables, replace [t_i^k: NonNegReal] with [$\widehat{t_i^k}$: SymbolicTimeout]

   For timed variables, replace [x_i^k: NonNegReal] with [$\widehat{x_i^k}$: SymbolicTimeData]

   Remove $clock_i$

   Add the following five (one local and four global) variables.
   global picked_ts_ID_set: Set of Natural := {}
   global picked_tn_ID_set: Set of Natural := {}
   global expired_tn_list: Set of Natural := {}
   global expired_ts_list: Set of Natural := {}
   expiration_check_vector: ExpirationCheckVector

5. [**Removing trajectory**]: Remove the trajectory definition from the process model.

6. [**Transitions**]

   For transitions, the abstraction algorithm is defined as a function Υ_α from the syntax defined in the template to the syntax in the ordinary I/O automaton programming language, where α is the supplemental information used for expiration checks that we discussed in Section 4.3.3. Given a part of the template T, we express that the abstraction changes T to T' by $\Upsilon_\alpha[\![T]\!] \rightarrow T'$.

   For a Boolean value *bool* and partial Tempo code $Code$, we use the notation $\langle Code \rangle^{bool}$ to express that if *bool* is true, then $\langle Code \rangle^{bool}$ is $Code$, else $\langle Code \rangle^{bool}$ is an empty sequence.

   We use the following auxiliary functions in the abstraction.

   Function new_time_nonce_ID is defined as follows:
   new_time_nonce_ID(picked_tn_ID_set: Set of Nat): Nat =
   min(Nat \ picked_tn_ID_set)

   Function one_new_tn_ID_picked is defined as follows:
   one_new_tn_ID_picked(picked_tn_ID_set: Set of Nat): Set of Nat =
   picked_tn_ID_set ∪ {new_time_nonce_ID(picked_tn_ID_set)}

   Functions new_time_stamp_ID and one_new_ts_ID_picked are defined similarly.

(a) **[Timeout actions]**

The precondition of a timeout action is over-approximated by $timeout\_is\_set_i^k$. In addition, expired_ts_list and expired_tn_list is updated when the timeout time is set with the slack value ε.

$\Upsilon_\alpha[\![$

---

pre $clock_i = t_i^k \wedge timeout\_is\_set_i^k$
eff $Assignments$

$]\!] \rightarrow$

---

pre $timeout\_is\_set_i^k$
eff if $\mathsf{slack\_added}(\widehat{t_i^k})$ then
$\mathsf{expired\_ts\_list} := \mathsf{expired\_ts\_list} \cup \mathsf{ts\_set}(\widehat{t_i^k})$;
$\mathsf{expired\_tn\_list} := \mathsf{expired\_tn\_list} \cup \mathsf{tn\_set}(\widehat{t_i^k})$; fi
$\Upsilon_\alpha[\![Assignments]\!]$

(b) **[Send and broadcast actions]**

The precondition of the action can possibly have the following two forms:

i. $ChooseTime \wedge ChooseInteractInst \wedge UntimedPred \wedge ExpirationCheck$, and
ii. $ChooseTime \wedge ChooseInteractInst \wedge UntimedPred$.

$ChooseTime$ has the following form: $\chi = TimeDataExpression$. If $TimeDataExpression$ uses a newly picked time nonce, we need to add the newly picked time nonce to picked_tn_ID_set in the abstracted version. Similarly, if $TimeDataExpression$ uses a newly picked time stamp (plus u), we need to add the newly picked time stamp to picked_ts_ID_set.

Effects of the action are defined using $Assignments$. In the following, we suppose that $Assignments$ consists of $Assignment_1 Assignment_2 \cdots Assignment_m$.

We have two abstractions depending on the form of precondition.

The easier case is that the precondition does not use an expiration check.

$\Upsilon_\alpha[\![$

---

pre $\chi = TimeDataExpression \wedge ChooseInteractInst \wedge UntimedPred$
eff $Assignments$

$]\!] \rightarrow$

---

pre $\widehat{\chi} = \Upsilon_\alpha[\![TimeDataExpression]\!] \wedge ChooseInteractInst \wedge UntimedPred$
eff $\Upsilon_\alpha[\![Assignments]\!]$;
$\langle \mathsf{picked\_tn\_ID\_set} := \mathsf{one\_new\_tn\_ID\_picked}(\mathsf{picked\_tn\_ID\_set})\rangle^{has\_new\_tn(TimeDataExpression)}$
$\langle \mathsf{picked\_ts\_ID\_set} := \mathsf{one\_new\_ts\_ID\_picked}(\mathsf{picked\_ts\_ID\_set})\rangle^{has\_new\_ts(TimeDataExpression)}$

---

where has_new_tn is a Boolean function that checks whether a given $TimeDataExpression$ uses new_time_nonce(), and has_new_ts is a Boolean function that checks whether a given $TimeDataExpression$ uses $\mathsf{clock_i} + \mathsf{u}$.

In the above abstraction, when sending time data χ uses a new time nonce and/or a new time stamp, book-keeping variables picked_tn_ID_set and picked_ts_ID_set are updated in the untimed version accordingly.

In the case that the precondition uses an expiration check, we need to add lines that conservatively approximates the expiration check. As we discussed in Section 4.3.3, we use a vector of Boolean values that represents the results of the expiration checks used in the underlying transition. In the cases for send and broadcast actions presented here, we use the transition parameter v of $\text{bcast}_i(J : \text{SetOfProcesses}, M : \text{UntimedMessage}, \widehat{\chi} : \text{SymbolicTimeData}, \kappa : \text{InteractInst}, v : \text{ExpirationCheckVector})$ and $\text{send}_i(j : \text{ProcessIndex}, M : \text{UntimedMessage}, \widehat{\chi} : \text{SymbolicTimeData}, \kappa : \text{InteractInst}, v : \text{ExpirationCheckVector})$. This v is used to bind the expiration check results in the precondition and then used in effects. Note that expiration_check_vector is assigned to v in the beginning of the effects in the following abstraction.

$\Upsilon_\alpha[\![$

---

pre $\chi = TimeDataExpression \wedge ChooseInteractInst \wedge UntimedPred \wedge ExpirationCheck$

eff $Assignments$

$]\!] \rightarrow$

---

pre $\widehat{\chi} = \Upsilon_\alpha[\![TimeDataExpression]\!] \wedge ChooseInteractInst \wedge UntimedPred \wedge$
expiration_consistent$(v) \wedge$
$v[\alpha(TimeDataExpression)] = \text{true}$

eff expiration_check_vector $:= v$
$\Upsilon_\alpha[\![Assignment_1]\!]\, \Upsilon_\alpha[\![Assignment_2]\!] \cdots \Upsilon_\alpha[\![Assignment_m]\!]$;
$\langle$picked_tn_ID_set := one_new_tn_ID_picked(picked_tn_ID_set);$\rangle^{has\_new\_tn(TimeDataExpression)}$
$\langle$picked_ts_ID_set := one_new_ts_ID_picked(picked_ts_ID_set);$\rangle^{has\_new\_ts(TimeDataExpression)}$

Note that in the above definition, $Assignments$ are divided into each assignment after the application of Υ_α. This is because for this untimed version of the transition, the process determines the results of expiration checks using the binding in the precondition, and thus should not perform the non-deterministic choice for the expiration checks in the effects, which we do for $\Upsilon_\alpha[\![Assignments]\!]$, as we describe later in this section.

(c) **[Internal actions]**

Internal actions have two types of conversions depending on whether they use an expiration check in the precondition or not.

$\Upsilon_\alpha[\![$

---

pre $UntimedPred \wedge ExpirationCheck$
eff $Assignments$

$]\!] \rightarrow$

---

pre $UntimedPred \wedge$
expiration_consistent$(v) \wedge$
$v[\alpha(TimeDataExpression)] = \text{true}$
eff expiration_check_vector $:= v$
$\Upsilon_\alpha[\![Assignment_1]\!] \, \Upsilon_\alpha[\![Assignment_2]\!] \cdots \Upsilon_\alpha[\![Assignment_m]\!]$;

$\Upsilon_\alpha[\![$

---

pre $UntimedPred$
eff $Assignments$

$]\!] \rightarrow$

---

pre $UntimedPred$
eff $\Upsilon_\alpha[\![Assignment_1]\!] \, \Upsilon_\alpha[\![Assignment_2]\!] \cdots \Upsilon_\alpha[\![Assignment_m]\!]$;

(d) [**Assignments**]

If $Assignments$ include an assignment that uses an expiration check, then we use the following conversion:

$\Upsilon_\alpha[\![Assignments]\!] \rightarrow$

---

expiration_check_vector $:=$ choose$\{v \mid$ expiration_consistent$(v)\}$
$\Upsilon_\alpha[\![Assignment_1]\!] \, \Upsilon_\alpha[\![Assignment_2]\!] \cdots \Upsilon_\alpha[\![Assignment_m]\!]$
expired_tn_ID_list $:=$ add_tn_IDs_expired_with_slack(expired_tn_ID_list, expiration_check_vector)
expired_ts_ID_list $:=$ add_ts_IDs_expired_with_slack(expired_ts_ID_list, expiration_check_vector)

---

where add_tn_IDs_expired_with_slack represents adding time nonce IDs that are considered globally expired from the results of expiration checks ($TimeDataExpression_m + \varepsilon \leq clock_i$ was true), and is defined as follows:

add_tn_IDs_expired_with_slack(expired_tn_ID_list: Set of Natural, v: ExpirationCheckVector):

Set of Natural $=$
expired_tn_ID_list $\cup$ tn_IDs_expired_with_slack(v)

where

tn_IDs_expired_with_slack(v: ExpirationCheckVector): Set of Natural $=$
$\bigcup\{$ts_set$(\overleftarrow{\alpha}(m)) \mid m \in ExpirationCheckIndexWithSlack \wedge v[m]\}$

add_ts_IDs_expired_with_slack is defined similarly for time stamp IDs.

If *Assignments* does not use an expiration check, then we use the following conversion: $\Upsilon_\alpha[\![\, Assignments]\!] \rightarrow \Upsilon_\alpha[\![\, Assignment_1]\!]\, \Upsilon_\alpha[\![\, Assignment_2]\!] \cdots \Upsilon_\alpha[\![\, Assignment_m]\!]$

(e) **[Assignment]**

Now we explain the Υ_α abstraction for each assignment:

i. $UntimedDataAssignment$:
$\Upsilon_\alpha[\![\, UntimedDataAssignment]\!] \rightarrow UntimedDataAssignment.$

ii. $TimeDataAssignment$:
$\Upsilon_\alpha[\![\, x_i^k := TimeDataExpression]\!] \rightarrow$

$\widehat{x_i^k} := \Upsilon_\alpha[\![\, TimeDataExpression]\!]$
$\langle$picked_tn_ID_set := one_new_tn_ID_picked(picked_tn_ID_set)$\rangle^{has\_new\_tn(TimeDataExpression)}$
$\langle$picked_ts_ID_set := one_new_ts_ID_picked(picked_ts_ID_set)$\rangle^{has\_new\_ts(TimeDataExpression)}$

---

iii. $TimeoutAssignment$:
When $TimeoutAssignment$ is derived into $t_i^k := TimeoutUpdateExpression;\ timeout\_is\_set_i^k := true;$, we have the following two cases depending on what $TimeoutUpdateExpression$ is derived into.

A. When $TimeoutUpdateExpression$ is derived into $TimeoutExpression$,
$\Upsilon_\alpha[\![\, t_i^k := TimeoutUpdateExpression;\ timeout\_is\_set_i^k := true;]\!] \rightarrow$

$\widehat{t_i^k} := \Upsilon_\alpha[\![\, TimeoutExpression]\!];\ timeout\_is\_set_i^k := true;$
$\langle$picked_tn_ID_set := one_new_tn_ID_picked(picked_tn_ID_set);$\rangle^{has\_new\_tn(TimeoutExpression)}$
$\langle$picked_ts_ID_set := one_new_ts_ID_picked(picked_ts_ID_set);$\rangle^{has\_new\_ts(TimeoutExpression)}$

B. When $TimeoutUpdateExpression$ is derived into $\max(t_i^{k_2}, TimeoutExpression)$,
$\Upsilon_\alpha[\![\, t_i^k := TimeoutUpdateExpression;\ timeout\_is\_set_i^k := true;]\!] \rightarrow$

$timeout\_consistency\_error := check\_timeout\_consistency(\widehat{t_i^{k_2}}, \Upsilon_\alpha[\![\, TimeoutExpression]\!]$
$\widehat{t_i^k} :=$ to_symb_max$(\widehat{t_i^{k_2}}, \Upsilon_\alpha[\![\, TimeoutExpression]\!]);\ timeout_is_set_i^k :=$ $true;$
$\langle$picked_tn_ID_set := one_new_tn_ID_picked(picked_tn_ID_set);$\rangle^{has\_new\_tn(TimeoutExpression)}$
$\langle$picked_ts_ID_set := one_new_ts_ID_picked(picked_ts_ID_set);$\rangle^{has\_new\_ts(TimeoutExpression)}$

---

where functions check_timeout_consistency and to_symb_max is defined as follows:

check_timeout_consistency(t_1, t_2: SymbolicTimeout) Boolean =
is_zero(t_1) $\vee$ is_zero(t_2) $\vee$
slack_added(t_1) = slack_added(t_2)

to_symb_max(timeout1, timeout2: SymbolicTimeout): SymbolicTimeout :=

set_timeout(symb_max(symb_max_part(timeout1), symb_max_part(timeout2)),
slack_added(timeout1))

Note that in to_symb_max, only the slack-added ($+\varepsilon$) information of the first symbolic timeout time is used to defines the "max"ed timeout time. The consistency of the slack-added information (whether both timeouts use the same slack-added condition) is checked by check_timeout_consistency.

$\Upsilon_\alpha[\![t_i^k := 0;\ timeout\_is\_set_i^k := false;]\!] \rightarrow$
$\widehat{t_i^k} := (\{\}, \{\}, false);\ timeout\_is\_set_i^k := false;$

(Recall that $(\{\}, \{\}, false)$ symbolically represents 0 timeout time.)

iv. $IfStatement$:

$\Upsilon_\alpha[\![$ if $IfCondition$ then $Assignments_1$ else $Assignments_2$ fi$]\!] \rightarrow$
if $\Upsilon_\alpha[\![IfCondition]\!]$ then $\Upsilon_\alpha[\![Assignments_1]\!]$ else $\Upsilon_\alpha[\![Assignments_2]\!]$ fi

$\Upsilon_\alpha[\![$ if $IfCondition$ then $Assignments_1$ fi $]\!] \rightarrow$
if $\Upsilon_\alpha[\![IfCondition]\!]$ then $\Upsilon_\alpha[\![Assignments_1]\!]$ fi

v. $IfCondition$:

$IfCondition$ can be derived into one of the following two: $UntimedPred$ and $ExpirationCheck$.
When $IfCondition$ is $UntimedPred$,
$\Upsilon_\alpha[\![IfCondition]\!] \rightarrow UntimedPred$
When $IfCondition$ is $ExpirationCheck$,
$\Upsilon_\alpha[\![IfCondition]\!] \rightarrow \Upsilon_\alpha[\![ExpirationCheck]\!]$

vi. $ExpicationCheck$:

For expiration checks, we have four types of abstractions depending on the types of the comparison used for the expiration checks.

A. When $ExpicationCheck$ is derived into $TimeDataExpretion > clock_i$:
$\Upsilon_\alpha[\![ExpirationCheck]\!] \rightarrow$ expiration_check_vector$[\alpha_0(TimeDataExpression)]$

B. When $ExpicationCheck$ is derived into $TimeDataExpretion + \varepsilon > clock_i$:
$\Upsilon_\alpha[\![ExpirationCheck]\!] \rightarrow$ expiration_check_vector$[\alpha_\varepsilon(TimeDataExpression)]$

C. When $ExpicationCheck$ is derived into $TimeDataExpretion \leq clock_i$:
$\Upsilon_\alpha[\![ExpirationCheck]\!] \rightarrow \neg$expiration_check_vector$[\alpha_0(TimeDataExpression)]$

D. When $ExpicationCheck$ is derived into $TimeDataExpretion + \varepsilon \leq clock_i$:
$\Upsilon_\alpha[\![ExpirationCheck]\!] \rightarrow \neg$expiration_check_vector$[\alpha_\varepsilon(TimeDataExpression)]$

vii. $TimeoutExpression$:

$TimeoutExpression$ can be derived into one of the four terms, $TimeDataExpression$, $TimeDataExpression+\varepsilon$, $clock_i+u+2\varepsilon$, and $\max(clock_i+u+2\varepsilon, TimeDataExpressioin+\varepsilon)$.

We use the function set_timeout defined as follows. This function converts symbolic time data to symbolic timeout time using the "slack-added" condition.

set_timeout(td: SymbolicTimeData, has_slack: Boolean): SymbolicTimeout
=
(tn_set(td), ts_set(td), has_slack)

A. When $TiemoutExpression$ is $TimeDataExpression$,
$\Upsilon_\alpha[\![TimeoutExpression]\!] \rightarrow$
$\mathsf{set\_timeout}(\Upsilon_\alpha[\![TimeDataExpression)]\!]$, false)

B. When $TiemoutExpression$ is $TimeDataExpression + \varepsilon$,
$\Upsilon_\alpha[\![TimeoutExpression]\!] \rightarrow$
$\mathsf{set\_timeout}(\Upsilon_\alpha[\![TimeDataExpression]\!]$, true)

C. When $TimeoutExpression$ is $clock_i + u + 2\varepsilon$,
$\Upsilon_\alpha[\![TimeoutExpression]\!] \rightarrow$
($\{\}$, $\mathsf{picked\_ts\_ID\_set}$, true)

D. When $TiemoutExpression$ is $\max(clock_i + u + 2\varepsilon, TimeDataExpressioin + \varepsilon)$,
$\Upsilon_\alpha[\![TimeoutExpression]\!] \rightarrow$
$\mathsf{set\_timeout}(\mathrm{symb\_max}((\{\}, \mathsf{picked\_ts\_ID\_set}),$
$\Upsilon_\alpha[\![TimeDataExpression]\!])$, true)

viii. $TimeDataExpression$:
$TimeDataExpression$ can be derived into one of the following five terms: $new\_time\_nonce()$, $clock_i + u$, x_i^k, χ, and $\max(TimeDataExpression_1, TimeDataExpression_2)$.

A.

B. When $TimeDataExpression$ is $new\_time\_nonce()$,
$\Upsilon_\alpha[\![TimeDataExpression]\!] \rightarrow (\{\mathsf{new\_tn\_ID}(\mathsf{picked\_tn\_ID\_set})\}, \{\})$

C. When $TimeDataExpression$ is $clock_i + u$,
$\Upsilon_\alpha[\![TimeDataExpression]\!] \rightarrow (\{\}, \{\mathsf{new\_ts\_ID}(\mathsf{picked\_ts\_ID\_set})\})$
When $TimeDataExpression$ is x_i^k,
$\Upsilon_\alpha[\![TimeDataExpression]\!] \rightarrow \widehat{x_i^k}$.

D. When $TimeDataExpression$ is χ:[9]
$\Upsilon_\alpha[\![TimeDataExpression]\!] \rightarrow \widehat{\chi}$.

E. When $TimeDataExpression$ is $\max(TimeDataExpression_1, TimeDataExpression_2)$:
$\Upsilon_\alpha[\![TimeDataExpression]\!] \rightarrow$
$\mathsf{smb\_max}(\Upsilon_\alpha[\![TimeDataExpression_1]\!], \Upsilon_\alpha[\![TimeDataExpression_2]\!])$.

Example 5. We show an example of time data abstraction. Figures 3 and 4 represent the untimed version of the resource-sharing Protocol 5. Observe that now we use symbolic time data and timeouts instead of numerical ones. Consistency of the timeout update is checked before updating timeouts using a "max" of two symbolic timeout times. This consistency check appears in the receive action of CONST, CUSTOM, and BROKEN. In the receive action of BROKEN, the time-stamp-estimation trick is approximated by ($\{\}$, $\mathsf{picked\_ts\_ID\_set}$, true). In this protocol, expiration checks of the form $\chi > clock_i$ or $\chi \leq clock_i$ are not used. Therefore, we do not actually use the expired ID lists. □

[9]This χ can be the sending time data in $\mathrm{bcast}(J, \chi, M, \kappa)$ or $\mathrm{send}(j, \chi, M, \kappa)$, or the received time data in $\mathrm{receive}(J, \chi, M, \kappa)$. For $\mathrm{bcast}(J, \chi, M, \kappa)$ and $\mathrm{send}(j, \chi, M, \kappa)$, this χ is bounded to some time data in the precondition of the actions.

Color = enumeration of 'RED', 'BLUE'
Message = enuemration of 'CUSTOM', 'CONST', 'ACK', 'BROKEN'
ProgramCounter = enumeration of 'idle', 'waiting_for_ack', 'sending_ack',
'waiting_for_timeout', 'crit_ready', 'crit'

Automaton INTIMED_Time_Sharing_Process$_i$(my_color: Color, opponent_group: Process_IDs)
signature
output send$_i$(j: Process_ID, msg: Message, time_data: SymbolicTimeData, κ: InteractInst)
output bcast$_i$(J: Process_IDs, msg: Message, time_data: SymbolicTimeData, κ: InteractInst)
input receive$_i$(j: Process_ID, msg: Message, time_data: SymbolicTimeData, κ: InteractInst)
output timeout$_i$
states
timeout_time: SymbolicTimeout := ({},{},false)
timeout_is_set: Boolean := false
pc: ProgramCounter := if my_color = RED then crit_ready else idle fi
my_interact_inst: InteractInst := $\perp$
ack_interact_inst: Array Process_ID of InteractInst := [for (i: Process_ID): $\perp$];
poposing_time_nonce: SymbolicTimeData := ({},{})
ack_rcvd_list: Set of Process_IDs := $\emptyset$
ack_sending_list: Set of Process_IDs := $\emptyset$
waiting_for_msg: Array Process_ID of Boolean := [for (i: Process_ID): true];
global picked_ts_ID_set: set of Natural := {};
global picked_tn_ID_set: set of Natural := {};
global expired_ts_ID_list: set of Natural := {};
global expired_tn_ID_list: set of Natural := {};

Figure 3: Time data abstraction of the resource sharing Protocol 5. Part 1 of 2

transitions

output bcast$_i$(opponent_group, 'CONST', $\widehat{ts\_u}$, $\perp$)
  pre $\widehat{ts\_u}$ =
    new_time_stamp_ID(picked_time_stamp_ID_set) $\wedge$
    pc = crit_ready
  eff pc := crit;
    timeout_time := $\widehat{ts\_u}$;
    timeout_is_set := true;

output bcast$_i$(opponent_group, 'CUSTOM', $\widehat{tn}$, κ)
  pre $\widehat{tn}$ =
    new_time_nonce_ID(picked_time_nonce_ID_set) $\wedge$
    pc = crit_ready $\wedge$
    κ = new_interact_inst()
  eff pc := waiting_for_ack
    proposing_time_nonce := $\widehat{tn}$
    my_interact_inst := κ

input receive$_i$(j, 'CONST', $\widehat{ts\_u}$, κ)
  eff expiration_vector :=
      choose{v : $ExpirationVector$ |
          expiration_consistent(v)}
    if pc $\neq$ crit then
    waiting_for_msg[j] := false;
    proposing_time_nonce := ({},{});
    ack_rcvd_list := $\emptyset$; %% resetting
    if ¬expiration_vector[1] then
      if pc $\neq$ sending_ack
        then pc := waiting_for_timeout fi;
      timeout_consistency_error =
        check_timeout_consistency(timeout_time,
          set_timeout($\widehat{ts\_u}$, true))
      timeout_time :=
        to_symb_max(timeout_time,
          set_timeout($\widehat{ts\_u}$, true));
      timeout_is_set := true; fi fi

input receive$_i$(j, 'CUSTOM', $\widehat{tn}$, κ)
  eff expiration_vector :=
      choose{v : $ExpirationVector$ |
          expiration_consistent(v)}
    if pc $\neq$ crit then
    waiting_for_msg[j] := false
    proposing_time_nonce := ({},{});
    ack_rcvd_list := $\emptyset$; %% resetting
    if ¬expiration_vector[1] then
      pc := sending_ack;
      ack_interact_inst[j] := κ;
      timeout_consistency_error =
        check_timeout_consistency(timeout_time,
          set_timeout($\widehat{tn}$, true))
      timeout_time :=
        to_symb_max(timeout_time,
          set_timeout($\widehat{tn}$, true));
      timeout_is_set := true;
      ack_sending_list := ack_sending_list $\cup$ j;
    fi fi

input receive$_i$(j, 'BROKEN', ({},{}), $\perp$)
  eff if (pc = idle $\vee$ pc = waiting_for_timeout $\vee$
        pc = sending_ack) then
      ack_rcvd_list := $\emptyset$; %% resetting
      waiting_for_msg[j] := false;
      if pc = idle
        then pc := waiting_for_timeout fi;
      timeout_consistency_error =
        check_timeout_consistency(timeout_time,
          ({}, picked_ts_ID_set, true)
      timeout_time :=
          to_symb_max(timeout_time,
            ({}, picked_ts_ID_set, true));
      timeout_is_set := true;
    fi;

output send$_i$(j, 'ACK', ({},{}), κ)
  pre pc = sending_ack $\wedge$ j $\in$ ack_sending_list $\wedge$
    κ = ack_interact_inst[j]
  eff ack_sending_list := ack_sending_list $\setminus$ j;

input receive$_i$(j, 'ACK', $\widehat{tn}$, κ)
  eff if pc = waiting_for_ack $\wedge$
        κ = my_interact_inst then
      ack_rcvd_list := ack_rcvd_list $\cup${j};
      if ack_rcvd_list = opponent_group
        then pc := crit;
          timeout_time :=
            proposing_time_nonce;
          timeout_is_set := true;
      fi fi

output timeout$_i$
  pre clock$_i$ = timeout_time $\wedge$ timeout_is_set
  eff timeout_time := ({}, {}, false);
    timeout_is_set := false;
    if pc = crit then
      pc := idle;
      for (j: opponent_group):
        waiting_for_msg[j] := true;
      ack_rcvd_list := $\emptyset$
      proposing_time_nonce := ({}, {})
    elseif (pc = waiting_for_timeout $\vee$
        pc = sending_ack) $\wedge$
      ($\forall$(j:opponent_group):¬waiting_for_msg[j]) then
      pc := crit_ready;
      ack_sending_list := $\emptyset$; fi

4.5 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 $timeout_k^i$ is set to a time larger than the time for another timeout $timeout_\ell^j$ by more than ε, then $timeout_k^i$ occurs after $timeout_\ell^j$ has done so. By rephrasing the above statement in symbolic representations, we obtain the following. Timeout action $timeout_k^i$ of a process i is disabled (even when $timeout\_is\_set_i^k$ is true) when the following condition holds:

$$\begin{aligned}\exists t_\ell^j \in Timeout\_variables_j : (j \neq i) \wedge \\ & (\mathsf{tn\_set}(t_\ell^j) \subseteq \mathsf{tn\_set}(t_k^i)) \wedge \\ & (\mathsf{ts\_set}(t_\ell^j) \subseteq \mathsf{ts\_set}(t_k^i)) \wedge \\ & \neg \mathsf{slack\_added}(t_\ell^j) \wedge \mathsf{slack\_added}(t_k^i)\end{aligned}$$

4.6 Time Data Reuse and Compression

The basic ideas of time data reuse and compression are the following two: 1. [Time Data Reuse] Reuse the time stamp and time nonce IDs that had been used, but are no longer used in the current state of an automata execution; and 2. [Time Data Compression] Represent by one time data ID a set of time data IDs of the untimed model after time data abstraction.

We also need to reuse interaction instances so that the abstracted model has the efficient use of interaction instances in the finite domain.

We present the above two techniques as follows. First, we define a new abstraction for the template by which the resulting model reuses time stamp and time nonce IDs in a way similar to time data abstraction presented in Section 4.4. Then, we *add new transitions* to the resulting model of the first step. These new transitions represent the *compression steps* by which multiple state variables of the composed system change at the same time in order to perform a compression of time data. By a compression of time data, a set of time stamp or time nonce IDs is symbolically represented by one time stamp or time nonce ID. Before going to the detailed explanation of how this compression works, we first show an example of a compression.

Example 6. Suppose the composed system consists of two processes P_1 and P_2 and channels between them. P_1 and P_2 has exactly one timed variable x_1 and x_2, respectively, and channels can store exactly one time data, stored in $cx_{1\to 2}$ and $cx_{2\to 1}$, respectively.

In addition, suppose that in the current state of the automaton x_1's symbolically represented time data is $(\{2,3,4\},\{\})$, x_2's time data is $(\{1,2,3\},\{\})$, $cx_{1\to 2}$ stores $(\{\},\{\})$, and $cx_{2\to 1}$ stores $(\{2,3\},\{\})$.

We can observe that for time stamp IDs, 2 and 3 are always used as a pair (or both are not used, in x_2). In such a case, we can actually compress the pair 2 and 3 in every timed variable in the system to 2 (or 3 or any other unused time stamp ID) without changing behavior of the automaton. This is because only the relative size of the time data affects the behavior of automata (determined by what transitions of which process are enabled at a particular state).

The resulting state has the following value assignment after the compression: $x_1 = (\{2, 4\}, \{\})$, $x_2 = (\{1, 2\}, \{\})$, $cx_{1\to 2} = (\{\}, \{\})$, and $cx_{2\to 1} = (\{2\}, \{\})$ – the time stamp ID 2 now symbolically represents the pair of IDs 2 and 3 before the compression. □

In the following Section 4.6.1, we present the abstraction for the template for reusing time stamp and time nonce IDs. In Section 4.6.2, we explain how a compression step is performed.

4.6.1 Reusing a Finite Number of Time Stamp and Time Nonce IDs and Interaction Instances

In this subsection, we present a new abstraction for the TIOA model described by the template for TO-abstraction. The basic structure of the abstraction function is very similar to the one for time data abstraction (TDA) discussed in 4.3. Therefore, we only show the parts that are difference from TDA.

The basic idea of this abstraction is, instead of using infinite time stamp and time nonce IDs as by TDA, we bound the number of time stamp and time nonce IDs that we can use for the abstraction, and reuse the time stamp and time nonce IDs that are no longer used in the current state. We use a similar "reuse" strategy for interaction instances as well.

In order to retrieve the relationship between this 'reuse' model and the untimed model by TDA, we retain auxiliary variables for TDA, such as new_tn and taken_ts_set, in this abstraction. Moreover, in order for a proof of simulation relation from the reuse version to the TDA version, we run the TDA version and the reuse model in parallel inside of the reuse abstraction.[10] We use a *mapping* variable that stores relationship between time stamp and time nonce IDs for the TDA version and those for this reuse version. The variable actually stores a pair of natural numbers. For example a mapping for time stamps may have pairs (1,3) and (2,16), which represents that ts_3 of the TDA version is represented by ID number 1 in the reuse version, and ts_{16} is represented by 2.

This mapping is not used to determine the behavior of the underlying automaton, but used just to keep information about the relationship between the TDA version and the reuse version. The information is useful for the reader to understand how the reuse version is related to the TDA version, and also for a proof of a simulation relation from the reuse version to the TDA version.

The parts of the definition of the reuse abstraction that are different from TDA is shown in the following.

1. Automaton parameters: Now we have the bounded size time stamp and time nonce ID space, represented by ts_size and tn_size, respectively. We also have the bounded interaction instance domain, $\{1, \cdots, intr\_inst\_size\}$.

[10] Since this parallel run is for a proof, but not for functioning the untimed abstraction, we can remove this TDA part in the actual implementation of the untimed model in a model-checker. Indeed, we took this approach in the DHCP Failover case study.

$Reuse\_P_i(ts\_size, tn\_size, intr\_inst\_size$: Natural, p : OtherDiscreteParameters)

2. Data types: The spaces (type) for time stamp and time nonce IDs are bounded by ts_size and tn_size, respectively.

   ReuseTimeStampID = $\{1, \cdots, ts\_size\ \}$
   ReuseTimeNonceID = $\{1, \cdots, tn\_size\ \}$
   ReuseInteractionInstances = $\{1, \cdots, intr\_inst\_size\ \}$

3. States: In addition to the auxiliary variables used for TDA (we retain these variables to keep information about the relationship between the TDA version and the reuse verion), we use eight new auxiliary variables.

   (a) picked_ts_in_use$_{REUSE}$ and picked_tn_in_use$_{REUSE}$ stores which time stamp and time nonce IDs are in use in the current state. These are global variables, and are defined as the set of time nonce IDs stored in timed and timeout variables in processes and channel buffers, $\bigcup\{\mathsf{tn\_set}(\widehat{t_i^k})\} \cup \bigcup\{\mathsf{tn\_set}(\widehat{x_i^k})\} \cup \bigcup\{\mathsf{tn\_set}(\widehat{cx_{i\to j}^k})\}$ and the set of time stamp IDs stored in timed and timeout variables in processes and channel buffers, $\bigcup\{\mathsf{ts\_set}(\widehat{t_i^k})\} \cup \bigcup\{\mathsf{ts\_set}(\widehat{x_i^k})\} \cup \bigcup\{\mathsf{ts\_set}(\widehat{cx_{i\to j}^k})\}$, respectively. In other words, picked_ts_in_use$_{REUSE}$ and picked_tn_in_use$_{REUSE}$ are referred as "aliases" of the above described sets of time nonce IDs and time stamp IDs. Therefore, when the values of $\{\widehat{t_i^k}\}$, $\{\widehat{x_i^k}\}$, or $\{\widehat{cx_{i\to j}^k}\}$ change, the values of picked_tn_in_use$_{REUSE}$ and picked_ts_in_use$_{REUSE}$ change automatically.

   We use a similar strategy for interaction instances: picked_interact_inst_in_use$_{REUSE}$ stores which interaction instances are in use in the current state (in state variables of processes and sending data in channels), and is defined as $\bigcup\{w_i^k\} \cup \bigcup\{cw_{i\to j}^k\}$

   (b) expired_ts_ID_list$_{REUSE}$ and expired_tn_ID_list$_{REUSE}$ are the reuse versions of expired_ts_ID_list and expired_tn_ID_list, respectively.

   (c) time_stamp_unavailable_error is set to true when all time stamp IDs in the bounded ID space are used when a process needs a new time stamp ID.

   time_nonce_unavailable_error and intr_inst_unavailable_error are used in the same way, but for time nonces and interaction instances, respectively.

   (d) reuse_ts_mapping represents a mapping from a time stamp ID in the reuse version to a time stamp ID in the TDA version. reuse_tn_mapping and reuse_intr_inst_mapping are used in the same way, but for time nonces and interaction instances, respectively.

   We conduct the following abstraction (code-to-code conversion) for the reuse case.

   CHANGE $\{\widehat{t_i^k}\}$, $\{\widehat{x_i^k}\}$ and $\{w_i^k\}$ to global variables (since these are accessed globally by using picked_tn_in_use$_{REUSE}$, picked_ts_in_use$_{REUSE}$, and picked_interact_inst_in_use$_{REUSE}$).

   ADD to the state variable declaration:

   ---

   global picked_tn_in_use$_{REUSE}$: Set of ReuseTimeNonceID
   %% defined as $\bigcup\{\mathsf{tn\_set}(\widehat{t_i^k})\} \cup \bigcup\{\mathsf{tn\_set}(\widehat{x_i^k})\} \cup \bigcup\{\mathsf{tn\_set}(\widehat{cx_{i\to j}^k})\}$

global $\mathsf{picked\_ts\_in\_use}_{REUSE}$: Set of ReuseTimeStampID

%% defined as $\bigcup\{\mathsf{ts\_set}(\widehat{t_i^k})\} \cup \bigcup\{\mathsf{ts\_set}(\widehat{x_i^k})\} \cup \bigcup\{\mathsf{ts\_set}(\widehat{cx_{i\to j}^k})\}$

global $\mathsf{picked\_interact\_inst\_in\_use}_{REUSE}$: Set of ReuseInteractionInstances

%% defined as $\bigcup\{w_i^k\} \cup \bigcup\{cw_{i\to j}^k\}$

global $\mathsf{expired\_ts\_ID\_list}_{REUSE}$: Set of ReuseTimeStampID := {}

global $\mathsf{expired\_tn\_ID\_list}_{REUSE}$: Set of ReuseTimeStampID := {}

global time_stamp_unavailable_error: Boolean := false;

global time_nonce_unavailable_error: Boolean := false;

global intr_inst_unavailable_error: Boolean := false;

%% The following two global variables are used to keep correspondence of

%% the time stamp and time nonce IDs used in the TDA version and

%% those in the reuse version.

global reuse_ts_mapping: Set of (ReuseTimeStampID × Natural)

global reuse_tn_mapping: Set of (ReuseTimeNonceID × Natural)

global reuse_intr_inst_mapping: Set of (ReuseInteractionInstances × Natural)

%% The following six variables are for the original time data abstraction.

global picked_ts_ID_list: Set of Natural := {}

global picked_tn_ID_list: Set of Natural := {}

global expired_tn_ID_list: Set of Natural := {}

global expired_ts_ID_list: Set of Natural := {}

%% expiration_check_vector is used in the same manner as in the TDA version.

expiration_check_vector: Boolean := false

---

4. Transition: Basically the main difference from TDA is that when a process needs a new time stamp or time nonce, instead of finding the minimum available ID in the set of ID ever picked, we find the minimum available time stamp or time nonce ID that is *non in use*, and use that ID.

   We use Υ^{REUSE} to represent the abstraction for the reuse version.

   We need the following auxiliary functions $\mathsf{new\_tn\_ID}_{REUSE}$ and $\mathsf{new\_ts\_ID}_{REUSE}$ to find the minimum available time stamp and time nonce. We use the default values for these functions so that the function is well defined even when there is no available value. We add an online checking of the availability of a new time nonce or stamp ID: if no ID is unavailable when a process needs a new ID, the system raises the error flag. This online check is used when model-checking the abstracted model. Namely, the properties we check is the original safety property plus the property that the system does not raise the flag in the current execution.

   $\mathsf{new\_tn\_ID}_{REUSE}$(picked_tn_in_use$_{REUSE}$: Set of ReuseTimeNonceID): ReuseTimeNonceID =

   min*({tn_ID: ReuseTimeNonceID | tn_ID $\notin$ picked_tn_in_use$_{REUSE}$})

   $\mathsf{new\_ts\_ID}_{REUSE}$(picked_ts_in_use$_{REUSE}$: Set of ReuseTimeStampID): ReuseTimeStampID =

$\min^*(\{\text{ts\_ID: ReuseTimeStampID} \mid \text{ts\_ID} \notin \text{picked\_ts\_in\_use}_{REUSE}\})$

where

$\min^*(S) = \min(S)$ if $S \neq \emptyset$,

$\min^*(S) = 1$ if $S = \emptyset$ (the default value).

We use a strategy similar to time data IDs for reusing interaction instances: We find the minimum available interaction instance in its bounded domain, and use it as a new interaction instance. We also use the online check for the availability of a new interaction instance.

$\mathsf{new\_interact\_inst}_{REUSE}(\text{picked\_intr\_inst\_in\_use}_{REUSE}$: Set of ReuseInteractionInstances):
ReuseInteractionInstances =

$\min^*(\{\text{intr\_inst: ReuseInteractionInstances} \mid \text{intr\_inst} \notin \text{picked\_intr\_inst\_in\_use}_{REUSE}$
})

(a) [Precondition of timeout actions]

The abstraction for timeout actions is basically the same as TDA.

---

pre $clock_i = t_i^k \wedge timeout\_is\_set_i^k$
eff $Assignments$

---

becomes

---

pre $timeout\_is\_set_i^k$
eff if $\mathsf{slack\_added}(t_i^k)$ then
$\mathsf{expired\_ts\_ID\_list}_{REUSE} := \mathsf{expired\_ts\_ID\_list}_{REUSE} \cup \mathsf{ts\_set}(t_i^k)$;
$\mathsf{expired\_tn\_ID\_list}_{REUSE} := \mathsf{expired\_tn\_ID\_list}_{REUSE} \cup \mathsf{tn\_set}(t_i^k)$; fi
%% The following two lines are for maintaining relationship
%% between the TDA version and the reuse version.
$\mathsf{expired\_ts\_ID\_list} := \mathsf{expired\_ts\_ID\_list} \cup map\_with\_default(\mathsf{reuse\_ts\_mapping}, (\mathsf{ts\_set}(t_i^k)))$
$\mathsf{expired\_tn\_ID\_list} := \mathsf{expired\_tn\_ID\_list} \cup map\_with\_default(\mathsf{reuse\_tn\_mapping}, (\mathsf{tn\_set}(t_i^k)))$
$\Upsilon^{REUSE}[\![Assignments]\!]$

---

where $map\_with\_default(\mathsf{reuse\_ts\_mapping}, (\mathsf{ts\_set}(t_i^k)))$ applies $\mathsf{reuse\_ts\_mapping}$ to every element of $\mathsf{ts\_set}(t_i^k)$ by regarding $\mathsf{reuse\_ts\_mapping}$ as a partial function, and if $\mathsf{reuse\_ts\_mapping}$ is not defined for the operand, then returns the default value 1.[11] $map\_with\_default(\mathsf{reuse\_tn\_mapping}, (\mathsf{tn\_set}(t_i^k)))$ is defined similarly.

(b) [Assignments]

If $Assignments$ include an assignment that uses an expiration check, then we use the following conversion:

$\Upsilon_\alpha^{REUSE}[\![Assignments]\!] \rightarrow$

---

[11] Though we define the default value of $map\_with\_default(\mathsf{reuse\_ts\_mapping}(\mathsf{ts\_set}(t_i^k)))$ to make the automaton definition well-defined, $\mathsf{reuse\_ts\_mapping}$ is always defined for $\mathsf{ts\_set}(t_i^k)$ in any reachable state of the automaton. We use this fact to prove a simulation relation from the reuse model to the TDA model.

expiration_check_vector := choose$\{v \mid$ expiration_consistent$(v)\}$
$\Upsilon_\alpha^{REUSE}[\![Assignment_1]\!] \Upsilon_\alpha^{REUSE}[\![Assignment_2]\!] \cdots \Upsilon_\alpha^{REUSE}[\![Assignment_m]\!]$
expired_tn_ID_list$_{REUSE}$:=
add_tn_IDs_expired_with_slack(expired_tn_ID_list$_{REUSE}$, expiration_check_vector)
expired_ts_ID_list$_{REUSE}$:=
add_ts_IDs_expired_with_slack(expired_ts_ID_list$_{REUSE}$, expiration_check_vector)
%% The following two lines are for maintaining relationship
%% between the TDA version and the reuse version.
Let v = expiration_check_vector in
expired_tn_ID_list := expired_tn_ID_list $\cup$
$map\_with\_default$(reuse_tn_mapping, tn_IDs_expired_with_slack(v));
expired_ts_ID_list := expired_ts_ID_list $\cup$
$map\_with\_default$(reuse_ts_mapping, ts_IDs_expired_with_slack(v));

---

where tn_IDs_expired_with_slack represents time nonce IDs that are considered globally expired from the results of expiration checks ($TimeDataExpression_m + \varepsilon \leq clock_i$ was true), and is the same function used for the TDA version. Also, add_tn_IDs_expired_with_slack represents adding time nonce IDs that are considered globally expired from the results of expiration checks, and is the same function used for the TDA version.

(c) [Expiration check]
We use expired_ts_ID_list$_{REUSE}$ and expired_tn_ID_list$_{REUSE}$ instead of expired_ts_ID_list and expired_tn_ID_list for expiration_consistent.

(d) [Picking a new time nonce]
The basic idea is the following: The automaton uses the minimum available unused ID. If all IDs are in use, it raises the error flag.
All expressions of the form new_tn_ID(picked_tn_ID_set) are replaced by new_tn_ID$_{REUSE}$(picked_tn_in_use$_{REUSE}$).
The following expression in the original Υ_α appears when picking a new time nonce (if $has\_new\_tn(TimeoutExpression)$ is true).
$\langle$picked_tn_ID_set := one_new_tn_ID_picked(picked_tn_ID_set);$\rangle^{has\_new\_tn(Expression)}$
We replace every occurrence of this expression in Υ_α with the following lines in Υ_α^{REUSE}:

$\langle$
%% Check the availability for a time nonce ID.
if picked_tn_in_use$_{REUSE}$ $= \{1, \cdots,$ tn_size$\}$ then time_nonce_unavailable_error := true; fi
Let new_ID_REUSE = new_tn_ID$_{REUSE}$(picked_tn_in_use$_{REUSE}$) in
Let new_ID_TDA =new_tn_ID(picked_tn_ID_set) in
%% Update the time nonce mapping.
%% (This operation is performed to maintain the relationship between
%% the REUSE version and the TDA version.)
reuse_tn_mapping := reuse_tn_mapping $\oplus$ (new_ID_REUSE, new_ID_TDA)

%% Remove the newly picked time nonce ID from the expired list.

expired_tn_ID_list$_{REUSE}$:= expired_tn_ID_list$_{REUSE}$ \ {new_ID_REUSE}

%% Finally, update the picked time nonce IDs for the TDA version.

picked_tn_ID_set := one_new_tn_ID_picked(picked_tn_ID_set);

$\rangle^{has\_new\_tn(Expression)}$

where $\oplus$ represents a mapping overwrite, defined as follows.

Let A be of type Set of (Domain$_1$ $\times$ Domain$_2$), and (x,y) is of type Domain$_1$ $\times$ Domain$_2$.

$$A \oplus (x, y) = (A \setminus \{(x, y')|y' \in \text{Domain}_2\}) \cup \{(x, y)\}.$$

Note that in the above conversion for $REUSE$, picked_tn_in_use$_{REUSE}$ is not explicitly updated. This is because in the $REUSE$ version, picked_tn_in_use$_{REUSE}$ is defined using timeout variables $\{t_i^k\}$ and timed variables $\{x_i^k\}$, and is implicitly updated when $\{t_i^k\}$ or $\{x_i^k\}$ changes their values.

(e) [Picking a new time stamp]

We use the same strategy as followed for time nonces when a process picks a new time stamp in this REUSE version. (When we replace "tn" with "ts" in the conversion for time nonces, we obtain the time stamp version.)

(f) [Picking a new interaction instance]

A process picks a new interaction instance using new_interact_inst() in the precondition of send and broadcast actions.

The modifications to the Υ_α for time data abstraction for the reuse version are the following:

REPLACE all new_interact_inst() WITH new_interact_inst_ID$_{REUSE}$(picked_intr_inst_in_use$_{REUSE}$.

ADD the following lines to the very end of the effects of send and broadcast actions if the precondition of the underlying send or broadcast action specifies that a process picks a new time nonce by κ = new_interact_inst().

%% Check the availability for a interaction instance.

if picked_intr_inst_in_use$_{REUSE}$ = {1, $\cdots$, intr_inst_size}

then intr_inst_unavailable_error := true; fi

%% Update the interaction instance mapping.

%% (This operation is performed to maintain the relationship between

%% the REUSE version and the TDA version.)

Let new_intr_inst_REUSE = new_interact_inst_ID$_{REUSE}$(picked_intr_inst_in_use$_{REUSE}$)

in

Let new_intr_inst_ORIGINAL = new_interact_inst() in

reuse_intr_inst_mapping := reuse_intr_inst_mapping $\oplus$ (new_intr_inst_REUSE, new_intr_inst_ORIGIN

(g) [Approximating time-stamp-estimation trick] In Inf, a time-stamp-estimation trick using $clock_i+u+2\varepsilon$ is approximated by using max(picked_ts_ID_set)$+u+\varepsilon$. In Fin, we only maintain picked_ts_in_use$_R EUSE$, the information of time stamps currently in use. Therefore, we approximate $clock_i+u+2\varepsilon$ using max(picked_ts_in_use$_{REUSE}$)$+u+$

ε, which is symbolically represented by ({},picked_ts_in_use$_{REUSE}$, true). Intuitively, using picked_ts_in_use$_{REUSE}$ instead of picked_ts_ID_set should not change the behavior of the model since $clock_i + \varepsilon$ represents an upper bound for the time stamps picked thus far, and what we especially care is that it is an upper bound for the time stamps currently in use. Actually, for soundness of the abstraction, we only need the fact that the symbolic value max(picked_ts_in_use$_{REUSE}$) $+ u + \varepsilon$ represents is less than or equal to max(picked_ts_ID_set)$+u+\varepsilon$, which is easy to assert. We prove the soundness of the reuse version in Section 5.2.

4.6.2 Compressing Time Stamp and Time Nonce IDs

In this subsection, we explain how one can add the *compression steps* to the transitions of the reuse abstraction explained in Section 4.6.1.

A compression step executes the following three sub-steps (as one atomic transition):

1. [Find a set of time stamps]:

   First, the composed system finds a set of time stamps and a set of time nonces that the system can compress (replace it with just one time stamp or time nonce ID) without changing the behavior of automata. We call these sets *clusters* This finding step is done by looking at the entire timed variables (including time data stored in channels).

2. [Compress all pieces of time data at the same time]:

   Next, the system compress all pieces of time data in the system using clusters found for time stamps and time nonces. This is done by representing the set of time stamps found by one representative time stamp in the set (we use the minimum ID in the set), and doing the same thing for time nonces.

3. [Adjust the time stamp and time nonce ID mappings]:

   After the compression, we must adjust the ID mappings between the TDA version and the reuse (+ compression) version. Since now one time stamp ID may represent a set of time stamps in the TDA version, we use Set of (ReuseTimeStampID *Times* (Set of Natural)) for the time stamp mapping. For example, (2, {4, 5}) means that 2 represents max(ts_4, ts_5).

We explain the above three steps in more details in the following.

Step 1. Find a set of time stamps.

The following *find_ts_cluster* function is used to find a set of time stamps of size compression_size that can be used for a compression. We call a set of time stamps found by *find_ts_cluster* a *ts-cluster*. time_data_vector represents the vector of the time-data values of timed and timeout variables of the entire systems (including time data stored in channels). For timeout variables, the Boolean variable for the ε slack is ignored in this function.

find_ts_cluster(cluster_size: Natural, time_data_vector: TimedValueVector):
Set of ReuseTimeStampID =
Iterate through every subset comp_candidate of ReuseTimeStampID that has
cluster_size many elements:

if ∀ symbolic_time_data ∈ time_data_vector:
comp_candidate ⊆ ts_set(symbolic_time_data) ∨
comp_candidate ∩ ts_set(symbolic_time_data) = ø
then: return comp_candidate % A cluster is found.
% No cluster for time stamps is found.
return {}

The above function *find_ts_cluster* finds such a set of time stamp IDs that for every time data that exists in the entire system, the time stamp IDs in the set are either used together (if an element of the set is used in one piece of time data, then all elements are used in that piece) or are not used at all. In other words, the time stamp IDs in the set are always used as one cluster.

Function *find_tn_cluster* is defined similarly for time nonces.

Example 7. Suppose the system has four pieces of symbolic time data, $(\{1,2,3,4\},\{\})$, $(\{2,3,4\},\{\})$, $(\{1\},\{\})$ and $(\{2,3,4,6,7\},\{\})$, in its current state. Recall that the first element of each piece of time data represents time stamps. A ts-cluster of size three for the system at this state is $\{2,3,4\}$, since the ID numbers 2, 3, and 4 are always used as one cluster (or are not used at all). Any subset of a ts-cluster is also a ts-cluster: in this case, for example, $\{2,4\}$ is also a ts-cluster.

On the other hand, $\{1,2,3\}$ is not a ts-cluster, since the third piece of time data uses 1, but neither 2 nor 3. □

find_ts_cluster finds a ts-cluster in its given size of the cluster. A cluster in any size can be used to compress the time data vector without changing the automaton behavior. To allow maximum generality, we suppose that the system non-deterministically choose the size of the cluster in this second sub-step.

Step 2. Compress all pieces of time data at the same time

Once a ts-cluster (or a tn-cluster) is found, we want to represent the cluster using just one time stamp (or time nonce) ID. This step is done by removing all time stamp IDs in the cluster from the time data vector, except for one representative. We use the minimum ID in the cluster as the representative for the cluster.

The actual compression for one time data is defined as follows:

compress_time_data(ts_cluster: Set of ReuseTimeStampID,
tn_cluster: Set of ReuseTimeNonceID,
time_data: SymbolicTimeData): SymbolicTimeData =
(*compress_ts*(ts_set(time_data), ts_cluster), *compress_tn*(tn_set(time_data), tn_cluster))

compress_ts(ts_ids: Set of ReuseTimeStampID,
ts_cluster: Set of ReuseTimeStampID): Set of ReuseTimeStampID =
if ts_cluster ≠ ø then
ts_ids \ {non_min_ts: ReuseTimeStampID |
non_min_ts ∈ ts_cluster ∧ non_min_ts ≠ min(ts_cluster) }
else ts_ids fi

Function *compress_tn* is defined similarly.

In this second sub-step, the system applies the above defined compression function to all time data in the system using the time stamp and time nonce ID clusters found in the first sub-step.

Step 3. Adjust the time stamp ID and time nonce ID mappings.

After applying the compression, we must adjust the time stamp and time nonce ID mappings. Since one time stamp ID (or time nonce ID) now may represent a set of time stamps (or time nonces) in the TDA version, we use Set of (ReuseTimeStampID × (Set of Natural)) for the time stamp ID mapping reuse_ts_mapping, and Set of (ReuseTimeNonceID × (Set of Natural)) for the time nonce ID mapping reuse_tn_mapping.

The adjustment for time stamps is performed by updating the mapping in the following way: remove the mapping for the ts-cluster used in the compression, and add a new pair of type ReuseTimeNonceID *Times* (Set of Natural) that represents the representative of the ts-cluster used (the minimum time stamp ID in the cluster) maps to all TDA-version IDs that the IDs in the compressed ts-cluster were mapping to.

This adjustment process is formally defined in the following function *ts_mapping_adjust*. A function for time nonces, *tn_mapping_adjust*, is similarly defined.

ts_mapping_adjust(reuse_ts_mapping: Set of (ReuseTimeStampID × (Set of Natural)),
ts_cluster: Set of ReuseTimeStampID):
Set of (ReuseTimeStampID × (Set of Natural)) =
(reuse_ts_mapping \ {(comp_ts, TDA_ts_set) ∈ reuse_ts_mapping | comp_ts ∈ ts_cluster}) ⊕
(min(ts_cluster),
⋃{TDA_ts_set | ∃ comp_ts: comp_ts ∈ ts_cluster ∧
(comp_ts, TDA_ts_set) ∈ reuse_ts_mapping})

[Modification for the management of the expired time stamp and time nonce ID list]: To add the compression steps explained above to the system, we also modify how we manage the expired time stamp and time nonce ID list of the TDA version. In the simple reuse version, we use the ID mapping from the reuse version to the TDA version in order to find out what IDs in the TDA version correspond to the expired IDs in the reuse version. Now one expired ID in the reuse version may correspond to a set of IDs in the TDA version. Therefore, when reuse IDs are expired in the reuse version, we add all TDA-version IDs that are in the sets mapped from the reuse IDs: $map\_with\_default(\text{reuse\_ts\_mapping}, (\text{ts\_set}(t_i^k)))$ is replaced by $\bigcup\{map\_with\_default(\text{reuse\_ts\_mapping}, (\text{ts\_set}(t_i^k)))\}$. A similar modification is needed for time nonces as well.

5 Soundness of Timeout Order Abstraction

In this section, we describe what kind of properties we can guarantee for the untimed model resulting from timeout order abstraction, and prove this "soundness" of TO-abstraction using automata-theoretic simulation relation techniques.

We split the soundness theorem into two parts: one for the model resulting from the first two steps of TO-abstraction – time data abstraction and timeout order constraint, and one for the model resulting from the all three steps of TO-abstraction, including time data compression. We call the former model Inf (infinite-state untimed abstraction) and the latter model Fin (finite-state untimed abstraction). We call the original TIOA model described by the template for TO-abstraction, $Orig$.

For Inf, we guarantee the following fact by proving a simulation relation from $Orig$ to Inf. For any timed execution α of $Orig$, there exists a corresponding execution β of Inf such that for every non-negative number m and every untimed variable v_i^k in the system, the following condition holds: In the m-th discrete-step states s_m of α and r_m of β (the states that are reached just after the m-th discrete transition in α and β, respectively), $s_m.v_i^k = r_m.v_i^k$.

From the above fact, we conclude that any safety properties of the original model regarding untimed variables can be verified by verifying the same safety property in the untimed model resulting from time data abstraction. We call safety properties of the above explained type *untimed safety properties*. Untimed safety properties includes, for example, no two processes enters a critical reason at the same time (mutual exclusion), or a particular untimed variable of every process is never assigned to a certain bad value.

For Fin, we can guarantee only a weaker fact than the one for Inf, because Fin uses only finite number of time stamp and time nonce IDs and it may run out of available IDs in an execution. Therefore, we guarantee the claim about corresponding executions in Fin (a claim similar to Inf) only up to the point in an execution at which the time stamp and time nonce unavailable errors have not occurred.

5.1 Simulation Relation from $Orig$ to Inf

The following Lemma 1 states basic facts about timeout deadlines of $Orig$: If a timeout is set, then the clock never evolve beyond the timeout time until the timeout has performed.

Lemma 1. *For any reachable state s of Orig, if $timeout\_is\_set_i^k$ is true, then the following inequality holds:*

$$clock_i \leq t_i^k.$$

Proof. By induction over the length of executions. Suppose the condition of the lemma holds in state s. We consider state s' that can be reached from one discrete transition or one trajectory from s.

We need to look at only transitions or trajectories that would affect $clock_i$, t_i^k, or $timeout\_is\_set_i^k$.

1. Transition that sets a timeout: Because of the constraint that we discussed at the end of Section 3.5.5, a process assigns a value larger than the current value of $clock_i$ to t_i^k. Thus the condition holds.

2. Transition that resets a timeout: When t_i^k is assigned to 0, $timeout\_is\_set_i^k$ is assigned to false at the same time. Thus the condition is vacuously true.

3. Trajectory: The condition holds from the induction hypothesis and the stop when condition of the trajectory, $\exists k : clock_i \geq t_i^k \wedge timeout\_is\_set_i^k$.

□

We need some modifications to $Orig$ in order to prove a simulation relation from $Orig$ to Inf. We explain these modifications in the following. The modifications does not change the behavior of $Orig$, but add some feature to store extra information about the relationship between the original numeric version of time data and the symbolic version that is used in Inf.

We basically need to emulate Inf in $Orig$, in parallel to an original execution of $Orig$. By doing this, we can closely look at how these two models are related. To do so, we modify the $Orig$ by using the same abstraction scheme used for time data abstraction. The difference is that we retain all the original preconditions and effects of the transitions, and embed only *effects* of the Inf into $Orig$ as follows.

For all timed and timeout variables $\{x_i^k\}$ and $\{t_i^\ell\}$ of process P_i in $Orig$, we add the untimed counterparts $\{\widehat{x_i^k}\}$ and $\{\widehat{t_i^\ell}\}$, respectively, to emulate the value changes of the symbolic timed variables in Inf. We can basically just add the effects of transitions of Inf to the original effects of $Orig$. One technicality here is that Inf may use non-deterministic choice for expiration checks. In the emulated version, we do not use non-deterministic choice, but instead use the actual expiration check of the original model: if received time data has expired in $Orig$, then the emulated version uses this fact to determine whether the time data in the symbolic version has expired or not.

To maintain a close correspondence between the symbolic time nonces and time stamps in the simulated Inf and the original numeric time nonces and time stamps in $Orig$, picked_tn_ID_set and picked_ts_ID_set stores not only the symbolic time nonce and time stamp IDs (respectively) ever picked, but also the value of corresponding numeric time nonces and time stamps in $Orig$. More specifically, when a time nonce is picked, we add to picked_tn_ID_set a pair (i, v) that encodes that the symbolic time nonce ID i represents time nonce tn_i that has the numeric value v. The time stamp version, picked_ts_ID_set, is similarly maintained. The two expired ID lists, expired_ts and expired_tn, also stores pairs in the form of (i, v) in the emulated version. We use the following two auxiliary functions to access to the IDs and the values, respectively, of picked_tn_ID_set, picked_ts_ID_set, expired_tn_ID_list and expired_ts_ID_list.

$$identifiers(ID\_list : \text{Set of (Natural} \times \text{NonNegReal)}) : \text{set of Natural} := \{i \mid (i, v) \in ID\_list\}$$

$$values(ID\_list : \text{Set of (Natural} \times \text{NonNegReal)}) : \text{set of NonNegReal} := \{v \mid (i, v) \in ID\_list\}$$

We call the above discussed modified $Orig$ model, $Orig^{Inf}$.

We also want $Orig^{Inf}$ to attach the symbolic time data along with the numerical data when a process of it sends or broadcasts a message with time data. By doing so, $Orig^{Inf}$ can emulate Inf that sends time data only using symbolic values.

Recall that we use $cv_{i\to j}^k$ to represent the untimed data contained in the message stored in the k-th entry of the message channel (FIFO queue) from process i to process j, and use $cx_{i\to j}^k$

to represent the time data contained in the message stored in the same entry. For emulation of Inf in $Orig^{Inf}$, we use $\widehat{cx^k_{i\to j}}$ to represent the symbolic time data sent along with the numeric time data stored in $cx^k_{i\to j}$.

The two main differences between the actual Inf automaton and the emulated version inside $Orig^{Inf}$ are as follows.

1. The emulated version emulates only the effects of transitions of Inf, and does not use the precondition of Inf to determine whether a transition is enabled or not. The enabled condition for $Orig^{Inf}$ (which is extended with the emulated version of Inf) is solely determined by the original model $Orig$. The enumerated version does not use the scheduler to constrain the enabled conditions of the transitions, as opposed to Inf.

2. Inf uses a non-deterministic choice for expiration checks of received time data. In the emulated version, an expiration check does not use a non-deterministic choice, but instead determines the results of the expiration checks in the calculation of symbolic values, using the original results of expiration checks in $Orig$.

For Inf, we need to add dummy local clocks $clock_i$. These clocks are added just to make the untimed model Inf comparable to $Orig$ in the simulation relation proof. The clocks are not actually used by the processes. Each clock has the following trajectory definition. Stop when clause is not used.

---

trajectory
  trajdef
    $\mathrm{d}(clock_i) > 0$

To prove a simulation relation from $Orig^{Inf}$ to Inf, we need to prove certain invariants of $Orig^{Inf}$.

We need the following auxiliary functions to define invariants. conc_tn_max converts the set of symbolic time nonce IDs in the emulated Inf in $Orig^{Inf}$ to the original value in the $Orig$, which is the maximum of the corresponding time nonces. conc_ts_max does the similar job for the time stamps. symb2conc_max computes the numerical value of symbolic time data.

$$\mathsf{conc\_tn\_max}(symb\_tn\_set : \mathsf{Set\ of\ Natural}, s : \mathsf{State\ of\ ModifiedOrig}) : \mathsf{NonNegReal} := \max(values(\{(i,v) \in s.\mathsf{picked\_tn\_ID\_set} \mid i \in symb\_tn\_set\}))$$

$$\mathsf{conc\_ts\_max}(symb\_ts\_set : \mathsf{Set\ of\ Nat}, s : \mathsf{State\ of\ ModifiedOrig}) : \mathsf{NonNegReal} := \max(values(\{(i,v) \in s.\mathsf{picked\_ts\_ID\_set} \mid i \in symb\_ts\_set\}))$$

$$\mathsf{symb2conc\_max}(x : \mathsf{Symbolic\_Time\_Data}, s : \mathsf{State\ of\ ModifiedOrig}) : \mathsf{NonNegReal} := \max(\mathsf{conc\_tn\_max}(\mathsf{tn\_set}(x), s), \mathsf{conc\_ts\_max}(\mathsf{ts\_set}(x), s) + u)$$

Lemma 4 states a close correspondence between the numerical values of time data and timeout values and their symbolic representations in the emulated Inf in $Orig^{Inf}$. The original

numerical version and the converted numerical value of the symbolic version represent the exactly same value, except for the timeout values with a slack ε. The inequality for the timeout values with ε comes from a conservative approximation of "$clock_i + u + 2\varepsilon$". Since we approximate this value using "max(picked_ts_ID_set) $+ u + \varepsilon$", the symbolic version represents the numeric value that does not exactly match the numeric value of "$clock_i + u + 2\varepsilon$". However, we will assert by proving a simulation relation from $Orig^{Inf}$ to Inf, we do not need the exact numeric-value matching for the timeout value, but an inequality ([the numeric value represented by the symbolic timeout time] $\leq$ [the original numeric timeout time]) is sufficient for the case of "slack-added" ($+\varepsilon$) timeouts to prove the simulation relation. This fact implies that approximation of "$clock_i + u + 2\varepsilon$" by "max(picked_ts_ID_set) $+ u + \varepsilon$" (which may have a smaller, but not larger numeric value than the original one) is indeed conservative.

To maintain the above described close correspondence, we need to assert that the timeout consistency error has not occurred to the underlying execution. Recall that the timeout consistency error occurs when a timeout time update is performed with a 'max' operation of two symbolic timeout times with inconsistent slack condition: one with the slack ($+\varepsilon$), and the other without the slack. This check is performed as a run-time check when a model-checking is performed for the abstraction. We focus only on timeout-consistency-error-free executions for verifications – a model described using template that causes the timeout consistency error is not a valid model to perform TO-abstraction.

To only focus on only timeout-consistency-error-free executions, we use the following definition of timeout-consistency-error-free (TCE-free) reachable states. We first explain a general b-free execution and a b-free reachable state for a state predicate b that implies an error state.

Definition 1. [b-free execution]: Consider a TIOA A, and a state predicate (the error predicate) b for the states of A. An execution α of A is a b-free execution if every state σ' after a discrete transition or a trajectory appearing in α, $b(\sigma')$ is false.

Definition 2. [b-free reachable state]: Consider a TIOA A, and a state predicate (the error predicate) b for the states of A. A state σ of A is b-free reachable state if there exists a b-free execution α of A such that σ is the last state of α.

Definition 3. [Timeout-consistency-error-free (TCE-free) reachable state]: Given a (composed) TIOA model $Orig$ described by the template for TO-abstraction, consider $Orig^{Inf}$, $Orig$ with the emulated Inf. A state σ of $Orig^{Inf}$ is a timeout-consistency-error-free (TCE-free) reachable state if there exists an execution α of S such that σ is the last state of α, and in every state after a discrete transition or a trajectory appearing in α, the value of timeout_consistency_error is false.

To prove Lemma 4, we need Lemma 3, and Lemma 3 needs Lemma 2.

Lemma 2. *For any TCE-free reachable state s of $Orig^{Inf}$, for any $(ID, ts) \in s.$picked_ts_ID_set, there exists some process P_i such that $s.clock_i \geq ts$*

Proof. We can prove this easily by induction over the length of executions. We use the fact that, when a time stamp ts is taken by process P_i, $clock_i = ts$, and the value of $clock_i$ evolves non-decreasingly. □

Lemma 3. *For any TCE-free reachable state s of $Orig^{Inf}$ and any process P_i, $s.clock_i + \varepsilon > \max(values(s.\mathsf{picked\_ts\_ID\_set}))$.*

Proof. The condition of the lemma is equivalent to:

$$\forall (ID, ts) \in s.\mathsf{picked\_ts\_ID\_set}, \forall clock_i \in ClockVariables : s.clock_i + \varepsilon > ts.$$

We prove this condition. Fact 1: From Lemma 2, there exists some process P_j such that $s.clock_j \geq ts$. Fact 2: From the loosely-synchronized assumption of processes, for any process P_i, $s.clock_i + \varepsilon > s.clock_j$. From Facts 1 and 2, we obtain the required condition. □

Lemma 4. *For any TCE-free reachable state s of $Orig^{Inf}$, the following three conditions holds.*

1. $\forall s.x_i^k \in Orig.TimedVariables : \ s.x_i^k = \mathsf{symb2conc\_max}(s.\widehat{x_i^k}, s)$
2. $\forall s.t_i^k \in Orig.TimeoutVariables :$

$$\mathsf{slack\_added}(s.\widehat{t_i^k}) \Rightarrow s.t_i^k \geq \mathsf{symb2conc\_max}(\mathsf{symb\_max\_part}(s.\widehat{t_i^k}), s) + \varepsilon \quad \wedge$$
$$\neg\mathsf{slack\_added}(s.\widehat{t_i^k}) \Rightarrow s.t_i^k = \mathsf{symb2conc\_max}(\mathsf{symb\_max\_part}(s.\widehat{t_i^k}), s)$$

3. $\forall s.cx_{i\to j}^k \in Orig.ChannelTimeData : \ s.cx_{i\to j}^k = \mathsf{symb2conc\_max}(s.\widehat{cx_{i\to j}^k}, s)$

Proof. By induction over the length of executions of $Orig^{Inf}$. In the initial state, all x_i^k, t_i^k, and $cx_{i\to j}^k$ are set to zero, and all $\widehat{x_i^k}$, $\widehat{t_i^k}$, and $\widehat{cx_{i\to j}^k}$ are set to {} (an empty set). Thus the condition holds.

Inductive step: Basically, a discrete transition of Inf (and thus also Inf part of $Orig^{Inf}$) has the same effects as the corresponding transition of $Orig$ (when converted by $\mathsf{symb2conc\_max}$). The tricky part is the approximation of $clock_i + u + 2\varepsilon$ by $\max(\{picked\_ts\_ID\_set\}) + u + \varepsilon$.[12] We can see the above fact by examining how time data abstraction is done for Inf to symbolically emulate $Orig$.

Now we consider a transition that uses a timeout variable assignment using $clock_i + u + 2\varepsilon$. The first and the third conditions of the lemma hold from the same reason that we described above for other transitions.

We look at the second condition. The second implication of the second condition (timeout value without a slack ε) holds from the same reason as well. We prove the first implication of the second condition (timeout value with a slack). Since the timeout assignments that do not use $clock_i + u + 2\varepsilon$ preserves the condition because of the same reason as above, we consider only the timeout assignment that use $clock_i + u + 2\varepsilon$. In the emulated Inf in $Orig^{Inf}$, $clock_i + u + 2\varepsilon$ is approximated by $(\{\}, identifiers(picked\_ts\_ID\_set), true)$. Thus, it is sufficient to prove that the numeric value of $\max(values(picked\_ts\_ID\_set)) + u + \varepsilon$ in $Orig^{Inf}$ is less than the numeric value of $clock_i + u + 2\varepsilon$. From Lemma 3, $s.clock_i + \varepsilon > \max(values(s.picked\_ts\_ID\_set))$. Therefore, $s.clock + u + 2\varepsilon = (s.clock_i + \varepsilon) + u + \varepsilon > \max(values(s.picked\_ts\_ID\_set)) + u + \varepsilon$, as required. □

[12] Another notable difference between Inf and $Orig$ is how expiration checks are performed. In $Orig$, processes use numeric values to conduct the "real" expiration checks. Whereas in Inf, expiration checks are performed by non-deterministic choice. In $Orig^{Inf}$, process use the results of "real" expiration checks performed in $Orig$ to determine the expiration checks in Inf-part of $Orig^{Inf}$. Therefore, the difference of expiration checks in Inf and $Orig$ is not a problem here. We consider this difference in the proof of the simulation relation, Theorem 7.

The following Lemma 5 states the key property of the expired time nonce list – if a time nonce ID tn is in the expired time nonce list, then the corresponding numeric value of tn is greater than the local clock value $clock_i$ of any process i. Lemma 6 states a similar property for the expired time stamp list.

Lemma 5. *For any TCE-free reachable state s of $Orig^{Inf}$, if a time nonce tn is included in the expired time nonce list* expired_tn_ID_list*, then*

$$\forall(clock_i : \ ClockVariables) : \ clock_i > tn$$

Proof. By induction over the length of executions. The condition holds in the initial states since expired_tn_ID_list is empty in the states.

Inductive step: Suppose the condition of the lemma holds in state s. We consider state s' that can be reached from one discrete transition or one trajectory from s.

We need to look at only transitions or trajectories that would affect $clock_i$ or $expired\_tn$.

1. Transition that adds a time nonce tn_{p_*} to expired_tn_ID_list: This transition must be one of the following two actions:

   (a) [Timeout action] A timeout action $timeout_i^k$ whose symbolic timeout value $\widehat{t_i^k}$ is of the following form adds a time nonce tn_{p_*} to expired_tn_ID_list.

   $$(\{p_1, p_2, \cdots, p_m\}, \{\ell_1, \ell_2, \cdots, \ell_r\}, true).$$

   where $p_* \in \{p_1, p_2, \cdots, p_m\}$.

   The above symbolic timeout value is converted to a numeric value in the following form:

   $$\max(tn_{p_1}, tn_{p_2}, \cdots, tn_{p_m}, ts_{\ell_1} + u, ts_{\ell_2} + u, \cdots, ts_{\ell_r} + u) + \varepsilon,$$

   where $tn_{p_*} \in \{tn_{p_1}, tn_{p_2}, \cdots, tn_{p_m}\}$.

   Therefore,

   $$\max(tn_{p_1}, tn_{p_2}, ..., tn_{p_m}, ts_{\ell_1} + u, ts_{\ell_2} + u, \cdots, ts_{\ell_r} + u) + \varepsilon \geq tn_{p_*} + \varepsilon.$$

   From the precondition of a timeout action, $clock_i = t_i^k$.

   From Lemma 4,

   $$t_i^k \geq \max(tn_{p_1}, tn_{p_2}, ..., tn_{p_m}, ts_{\ell_1} + u, ts_{\ell_2} + u, ..., ts_{\ell_r} + u) + \varepsilon.$$

   Hence, $clock_i \geq tn_{p_*} + \varepsilon$. From the loosely-synchronized assumption of processes, for any process P_j, $clock_j + \varepsilon > clock_i$. Therefore, $clock_j + \varepsilon > clock_i \geq tn_{p_*} + \varepsilon$, and thus $clock_j > tn_{p_*}$, as required for tn_{p_*}. The remaining time nonces in expired_tn_ID_list satisfy the condition because of the induction hypothesis and the fact that the transition does not affect $clock_i$.

   (b) [Transition with expiration checks]: A transition in which expiration checks for the slack-added values results in being true add expired time data ID to the expired ID list.

In Inf-part of $Orig^{Inf}$, the results of all expiration checks does not use non-deterministic choices, but the real results from $Orig$. Therefore, for time data τ, if $\tau + \varepsilon \leq clock_i$ is true at the current state, and τ is constructed by a symbolic 'max' of time data IDs that includes tn_{p_*}, $tn_{p_*} + \varepsilon \leq clock_i$. Therefore, by the same discussion for the case of the timeout action described above, $tn_{p_*} < clock_j$ for any local clock $clock_j$.

2. Trajectory: Since a trajectory does not change expired_tn_ID_list, and $clock_i$ evolves non-decreasingly, the induction hypothesis in s implies that the condition also holds in s'.

□

Lemma 6. *For any TCE-free reachable state s of $Orig$, if a time nonce ts is included in the expired time stamp list* expired_ts_ID_list*, then*

$$\forall(clock_i : \ ClockVariables) : \ clock_i > ts + u$$

Proof. We can prove this lemma in a way similar to Lemma 5. □

Now we are ready to prove a simulation relation from $Orig^{Inf}$ to Inf. The key to prove this simulation relation are the following two.

1. Examine that the timeout order constraint imposed by the scheduler in the Inf model conservatively approximates the actual timeout order that occurs in $Orig$. To do so, we use Lemma 4 that states the correspondence between pieces of symbolic time data in the emulated Inf and numerical values of them in $Orig^{Inf}$.

2. Examine that if statement branch and non-deterministic choice used in a transition using an expiration check in Inf conservatively approximates the effects of the corresponding original transition of $Orig$. To do so, we use Lemmas 5 and 6 that state the key properties of the expired time nonce list and the expired time stamp list emulated in $Orig^{Inf}$.

We use the following relation $\sim$ between states of $Orig^{Inf}$ and Inf as a candidate for a simulation relation from $Orig^{Inf}$ to Inf.

For states s and r of $Orig^{Inf}$ and Inf, respectively, $s \sim r$ iff all of the following conditions hold

1. $\forall s.v_i^k \in Orig^{Inf}.UntimedVariables : \ s.v_i^k = r.v_i^k$
2. $\forall s.x_i^k \in Orig^{Inf}.TimedVariables : \ s.\widehat{x_i^k} = r.\widehat{x_i^k}$
3. $\forall s.t_i^k \in Orig^{Inf}.TimeoutVariables : \ s.\widehat{t_i^k} = r.\widehat{t_i^k}$
4. $\forall s.timeout\_is\_set_i^k \in Orig^{Inf}.TimeoutSettingBooleans :$
   $s.timeout\_is\_set_i^k = r.timeout\_is\_set_i^k$
5. $\forall s.w_i^k \in Orig^{Inf}.InteractionInstances : \ s.w_i^k = r.w_i^k$
6. $\forall s.clock_i : Orig^{Inf}.ClockVariables : \ s.clock_i = r.clock_i$

7. $identifiers(s.\mathsf{picked\_tn\_ID\_set}) = r.\mathsf{picked\_tn\_ID\_set} \ \wedge$
   $identifiers(s.\mathsf{picked\_ts\_ID\_set}) = r.\mathsf{picked\_ts\_ID\_set}$

8. $identifiers(s.\mathsf{expired\_tn\_ID\_list}) = r.\mathsf{expired\_tn\_ID\_list} \ \wedge$
   $identifiers(s.\mathsf{expired\_ts\_ID\_list}) = r.\mathsf{expired\_ts\_ID\_list}$

9. $s.\mathsf{picked\_interaction\_instances} = r.\mathsf{picked\_interaction\_instances}$

10. $\forall s.cv^k_{i\to j} \in ChannelUntimedData: \ s.cv^k_{i\to j} = r.cv^k_{i\to j}$

11. $\forall s.cx^k_{i\to j} \in ChannelTimeData: \ \widehat{s.cx^k_{i\to j}} = \widehat{r.cx^k_{i\to j}}$

12. $\forall s.cw^k_{i\to j} \in ChannelInteractionInstance: \ s.cw^k_{i\to j} = r.cw^k_{i\to j}$

13. $s.\mathsf{timeout\_consistency\_error} = r.\mathsf{timeout\_consistency\_error}$

For invariants of $Orig^{Inf}$, we used the notion of timeout-consistency-error-free (TCE-free) executions and reachable states. we need to define a similar notion for simulation relations as well.

Definition 4. Consider two TIOA A and B. Let b be a pair of two state predicates b_A and b_B for states of A and B, respectively. A relation R between two sets of states of A and B is a b-free simulation relation if the following two condition holds:

1. [Initial condition]: If s_0 is an initial state of A, then $b_A(s_0)$ is false and there exists an initial state r_0 of B such that $b_B(s_0)$ is false and $s_0 \ R \ r_0$.

2. [Induction step]: For a b_A-free reachable state s of A and a b_B-free reachable state r of B, If $s \ R \ r$ and α is a step (either a discrete transition or a closed trajectory) of A starting from s, then there exists an execution fragment β of B such that β starts from r, and for s' and r' that represent the states reached after α and β, respectively, if $b_B(r')$ is false, then $s' \ R \ r'$.

3. [Error-fee implication]: For a state s of A and a state r of B, If $s \ R \ r$ and $b_B(r)$ is false, then $b_A(s)$ is false.

We use the state predicates "$\mathsf{timeout\_consistency\_error} = \text{true}$" to define a TCE-free simulation relation from $Orig^{Inf}$ to Inf.

Theorem 7. *The relation $\sim$ defined above is a TCE-free simulation relation from $Orig^{Inf}$ to Inf.*

Proof. Let s be a state of $Orig^{Inf}$ and r be a state of Inf such that $s \sim r$.

The initial condition and the error-free implication is easy.

We prove the step condition of the simulation relation in the following.

[Transitions] Suppose an action a of the original TIOA model is enabled in s. As we described earlier, the keys to the proof are the differences of when timeout actions are performed (in Inf, the scheduler constraints the timeout actions), and how expiration checks are evaluated (in Inf, non-deterministic choice under a certain constraint is used for expiration checks).

1. a uses expiration checks in its effects:
   We need to examine that an expiration check in Inf using constrained non-deterministic choice conservatively emulates the corresponding expiration check in $Orig$ (and also in $Orig^{Inf}$). Namely, we need to find for action a in $Orig$, the corresponding action b that have the exactly same results for the expiration checks. If the results of the checks are the same, then the results of symbolic time data manipulation in $Orig^{Inf}$ and Inf must be the same, and the same time data IDs must be added to the expired time data list.

   We find such an action b in Inf by analyzing the constraint for the expiration check vector imposed by expiration_consistent.

   We described two different versions of expiration_consistent in Section 4.3.3. The first version use only information of the expired ID lists to constraint the expiration check vector. Another version uses, in addition to the expired ID lists, the observation that one expiration check implies another expiration check in some cases. In both cases, the crucial point to examine for the proof is that the constraint for non-deterministic choice using the expired ID lists is conservative: If all time data IDs consisting symbolic time data $\widehat{\tau}$ are included in the expired ID lists, then the check corresponding to $\tau > clock_i$ must be false in the non-deterministic choice.

   From Lemmas 5 and 6, when the result of $\tau > clock_i$ is constrained to be false in Inf, every time nonce and time stamp ID that consists of that time data represents a numeric value smaller than any local clock value. This implies that $clock_i > symb2conc\_max(\widehat{\tau})$ in $Orig^{Inf}$. From Lemma 4, $symb2conc\_max(\widehat{\tau})$ in $Orig^{Inf}$ has the same numeric value as the original τ in $Orig^{Inf}$. Therefore, when the result of $\tau > clock_i$ is constrained to be false in Inf, $\tau > clock_i$ is indeed false in $Orig^{Inf}$. Therefore, we can always find an action b of Inf corresponding to a with respect to the results of expiration checks.

2. a uses expiration checks in the precondition and effects:
   When an expiration check is used in the precondition, we need to assert that the corresponding action b in Inf is also enabled. To assert the above described fact, we need to ensure that whenever the expiration check in a' precondition is evaluated to true, the expiration check in b' precondition must be evaluated to true as well. This holds from the fact that, as we described earlier for an action that uses expiration checks in its effects, the constraint for the expiration vector is conservative, and the expiration check vector that represents the same results of expiration checks in $Orig^{Inf}$ satisfies this constraint. For the effects of the action, we can use the same argument as described earlier.

3. $a = timeout_i^k$:
   Since a is enabled in s, $s.clock_i = s.t_i^k$ and $s.timeout\_is\_set_i^k$ is true.

   (a) We first prove that $timeout_i^k$ is also enabled in r. Since $s \sim r$ and $s.timeout\_is\_set_i^k$ is true, $r.timeout\_is\_set_i^k$ is also true. Therefore, it is sufficient to prove that $timeout_i^k$ is not disabled by the scheduler.

   Suppose, for contradiction, $timeout_i^k$ is disabled by the scheduler. This implies that the following condition holds: $\exists k', i' : tn(r.t_i^k) \supseteq tn(r.t_{i'}^{k'}) \wedge ts(r.t_i^k) \supseteq ts(r.t_{i'}^{k'}) \wedge$ slack_added$(r.t_i^k) \wedge \neg$slack_added$(r.t_{i'}^{k'})$.

From the above fact and $s \sim r$, $s.t_i^k \geq \mathsf{symb2conc\_max}(\mathsf{symb\_max\_part}(r.t_i^k), s) + \varepsilon$, $s.t_{i'}^{k'} = \mathsf{symb2conc\_max}(\mathsf{symb\_max\_part}(r.t_{i'}^{k'}), s)$, and $\mathsf{symb2conc\_max}(\mathsf{symb\_max\_part}(r.t_i^k), s) \geq \mathsf{symb2conc\_max}(\mathsf{symb\_max\_part}(r.t_{i'}^{k'}), s)$. Therefore, $s.t_i^k \geq s.t_{i'}^{k'} + \varepsilon$. Since $s.clock_i = s.t_i^k$, $s.clock_i \geq s.t_{i'}^{k'} + \varepsilon$. Hence, (i). $s.clock_i - \varepsilon \geq s.t_{i'}^{k'}$. From the loosely-synchronized assumption, (ii). $s.clock_{i'} > s.clock_i - \varepsilon$. From (i) and (ii), $s.clock_{i'} > s.t_{i'}^{k'}$. This contradicts Lemma 1 (which also holds in $Orig^{Inf}$).

(b) Now we prove that the effects of the transition a in $Orig^{Inf}$ and Inf preserves the relation $\sim$. This condition comes from the fact that a in the enumerated Inf in $Orig^{Inf}$ has the exactly same effects as those of a in Inf, except for a transition with an expiration check. For expiration checks used in timeout actions, we can use the same argument that we described earlier.

4. The proof for other transitions is easy considering that the emulated Inf in $Orig^{Inf}$ emulates Inf in the exactly same manner in these transitions.

[Trajectories] For a trajectory f of $Orig^{Inf}$, we can simply use the exactly same trajectory f to Inf, since the trajectory definition of Inf does not have any stopping condition and the evolve clause is same as the one for $Orig^{Inf}$. □

The following Corollary 8 can be easily proved using the simulation relation result proved as Theorem 7. This corollary implies that if an untimed safety property (a safety property that just refers to untimed variables) holds in Inf, then the same property holds also in $Orig^{Inf}$. Because the evolutions of untimed variables in $Orig^{Inf}$ and $Orig$ are exactly same, the above fact gives us the soundness of the abstraction used for Inf.

Corollary 8. *Suppose all executions of Inf that have less than k-many discrete transitions are TCE-free. We call this condition the depth-k global TCE-fee condition in Inf. Under this condition, all executions of $Orig^{Inf}$ that have less than k-many discrete transitions are also TCE-free, and for any execution α of $Orig^{Inf}$ that has less than k-many discrete transitions, there exists a TCE-free execution β of Inf such that for any $k \geq 0$, $\alpha|_k \sim \beta|_k$, where $\alpha|_k$ is the state just after the k-th appearing discrete action in α (when $k = 0$, the initial state), $\beta|_k$ is the state just after the k-th appearing discrete action in β (when $k = 0$, the initial state), and $\sim$ is the relation defined earlier in this section.*

Proof. Given an execution α of $Orig^{Inf}$ that has less than k-many discrete transitions, we can inductively construct a TCE-free execution β_k of Inf with k' ($k' \leq k$) discrete actions such that the last state of β'_k and $\alpha|'_k$ relates with respect to $\sim$. For the initial state α_1, we can use the initial condition of the simulation relation to find a state of Inf that is related to α_1. From the depth-k global TCE-fee condition in Inf and error-free-implication condition of $\sim$, α_1 is TCE-free. Now given that we construct a TCE-fee execution β'_k such that $k' < k$ $\beta'_k|_\ell \sim \alpha|_\ell$ for all $\ell, 0 \leq \ell \leq k'$. Let f be the trajectory and a be the discrete transition that appear between $\alpha|'_k$ and $\alpha|_{k'+1}$. Now by using the step condition of the simulation relation, we can find the corresponding valid pair of a trajectory g and a discrete transition b of Inf such that the state reached by g and then b from $\beta'_k|'_k$ is related to $\alpha|_{k'+1}$ with respect to $\sim$. Let the execution β'_k extended with g and then b, $\beta_{k'+1}$. Since $k' + 1 \leq k$, from the depth-k global TCE-fee condition

in Inf and error-free-implication condition of $\sim$, $\alpha|_{k'+1}$ is TCE-free. By repeating the above process, we can assert that α is TCE-free, and also create β that satisfies the condition of the corollary. □

5.2 Simulation Relation from Inf to Fin

Proving a simulation relation from Inf to Fin is relatively easy compared to that from $Orig^{Inf}$ to Inf. Basically, we can find the exact correspondence between the two models using the ID maps maintained in FIN that maps time data IDs used in Fin to those in Inf. The key technical differences between the two models occurs (as the reader may imagine) from the fact that in Inf, the model reuses IDs, and thus ID lists (the picked ID lists and the expired ID lists) maintained by Inf have only the information of the IDs currently in use, and not the whole history. Therefore, for these lists, we relate a state r in Inf to a state q in Fin if the set of IDs in the lists in q are a subset of the set of IDs in r. Since the approximation of the time-stamp-estimation trick in Fin uses $\mathsf{picked\_ts\_in\_use}_{REUSE}$, instead by $\mathsf{picked\_ts\_ID\_list}$ like in Inf, timeout times constructed using the time-stamp-estimation trick have different ID sets – the time stamp ID set used in Fin is a subset of those in Inf.

Fin actually behaves in a less constrained manner than Inf from the above described differences. This is because the model has less constraint for expiration checks when a less number of expired IDs are listed, and has less constraint for when timeout actions can be performed, when a slack-added symbolic timeout time has less number of IDs.

We need the following functions $map\_td\_with\_default$ and $map\_to\_with\_default$ for applying the mapping information of IDs for time data and timeout times.

$map\_td\_with\_default$(tn_map: Set of (Natural × Natural), ts_map: Set of (Natural × Natural),

td: SymbolicTimeData): SymbolicTimeData =

($map\_with\_default$(tn_map, $\mathsf{tn\_set}$(td)), $map\_with\_default$(ts_map, $\mathsf{ts\_set}$(td))

$map\_to\_with\_default$(tn_map: Set of (Natural × Natural), ts_map: Set of (Natural × Natural),

to: SymbolicTimeout): SymbolicTimeout =

($map\_with\_default$(tn_map, $\mathsf{tn\_set}$(to)), $map\_with\_default$(ts_map, $\mathsf{ts\_set}$(to), $\mathsf{slack\_added}$(to))

We use the following relation $\sim_*$ for a simulation relation from Inf to Fin.

Given a state r of Inf and a state q of Fin, r and q are related by $\sim_*$ ($r \sim_* q$), if all of the following conditions hold.

1. $\forall r.v_i^k \in Inf.UntimedVariables : \ r.v_i^k = q.v_i^k$
2. $\forall r.\widehat{x_i^k} \in Inf.SymbolicTimedVariables :$
   $r.\widehat{x_i^k} = map\_td\_with\_default(\mathsf{reuse\_tn\_mapping}, \mathsf{reuse\_ts\_mapping}, q.\widehat{x_i^k})$
3. $\forall r.\widehat{t_i^k} \in Inf.SymbolicTimeoutVariables :$

$$\mathsf{slack\_added}(r.\widehat{t_i^k}) \Rightarrow r.\widehat{t_i^k} \succeq map\_to\_with\_default(\mathsf{reuse\_tn\_mapping}, \mathsf{reuse\_ts\_mapping}, q.\widehat{t_i^k}) \wedge$$
$$\neg\mathsf{slack\_added}(r.\widehat{t_i^k}) \Rightarrow r.\widehat{t_i^k} = map\_to\_with\_default(\mathsf{reuse\_tn\_mapping}, \mathsf{reuse\_ts\_mapping}, q.\widehat{t_i^k})$$

4. $\forall r.timeout\_is\_set_i^k \in Inf.TimeoutSettingBooleans$:
   $r.timeout\_is\_set_i^k = q.timeout\_is\_set_i^k$

5. $\forall r.w_i^k \in Inf.InteractionInstances$: $r.w_i^k = map\_with\_default(\mathsf{reuse\_intr\_inst\_mapping}, q.w_i^k)$

6. $r.\mathsf{picked\_tn\_ID\_set} \supseteq map\_with\_default(\mathsf{reuse\_tn\_mapping}, q.\mathsf{picked\_tn\_in\_use}_{REUSE}) \wedge$
   $r.\mathsf{picked\_ts\_ID\_set} \supseteq map\_with\_default(\mathsf{reuse\_ts\_mapping}, q.\mathsf{picked\_ts\_in\_use}_{REUSE})$

7. $r.\mathsf{expired\_tn\_ID\_list} \supseteq map\_with\_default(\mathsf{reuse\_tn\_mapping}, q.\mathsf{expired\_tn\_ID\_list}_{REUSE}) \wedge$
   $r.\mathsf{expired\_ts\_ID\_list} \supseteq map\_with\_default(\mathsf{reuse\_ts\_mapping}, q.\mathsf{expired\_ts\_ID\_list}_{REUSE})$

8. $r.\mathsf{picked\_interaction\_instances} \supseteq map\_with\_default(\mathsf{reuse\_intr\_inst\_mapping}, q.\mathsf{picked\_interaction\_instances})$

9. $\forall r.cv_{i\to j}^k \in ChannelUntimedData$: $r.cv_{i\to j}^k = q.cv_{i\to j}^k$

10. $\forall r.cx_{i\to j}^k \in ChannelSymbolicTimeData$:
    $\widehat{r.cx_{i\to j}^k} = map\_td\_with\_default(\mathsf{reuse\_tn\_mapping}, \mathsf{reuse\_ts\_mapping}, \widehat{q.cx_{i\to j}^k})$

11. $\forall r.cw_{i\to j}^k \in ChannelInteractionInstance$:
    $r.cw_{i\to j}^k = map\_with\_default(\mathsf{reuse\_intr\_inst\_mapping}, q.cw_{i\to j}^k)$

12. $r.\mathsf{timeout\_consistency\_error} = q.\mathsf{timeout\_consistency\_error}$

Since Fin uses a runtime check of the availability of new time data IDs and interaction instances, in addition to a runtime check of timeout consistency, we use the following conditions for error-free executions of Fin.

1. [Time-consistency-error-free (TCE-free)]: In any state appearing in the underlying execution, the following condition does not hold: $\mathsf{timeout\_consistency\_error} = \mathrm{true}$

2. [Time-nonce-unavailable-error-fee (TNU-free)]: In any state appearing in the underlying execution, the following condition does not hold: $\mathsf{time\_nonce\_unavailable\_error} = \mathrm{true}$

3. [Time-stamp-unavailable-error-fee (TSU-free)]: In any state appearing in the underlying execution, the following condition does not hold: $\mathsf{time\_stamp\_unavailable\_error} = \mathrm{true}$

4. [Interaction-instance-unavailable-error-free (IIU-free)]: In any state appearing in the underlying execution, the following condition does not hold: $\mathsf{intr\_inst\_unavailable\_error} = \mathrm{true}$

We use the term TCE-TNU-TSU-IIU-free for an execution of Fin if the execution satisfies the above described four conditions, and TCE-TNU-TSU-IIU-free reachable states are defined as the last states of TCE-TNU-TSU-IIU-free executions.

We need the following lemma that states that a mapping from an ID or interaction instance of Fin to that of Inf is always defined for IDs and interaction instances currently in use.

Lemma 9. *For any TCE-TNU-TSU-IIU-free reachable state r of Fin, the following three conditions hold:*

1. *For all $tn \in r.\mathsf{picked\_tn\_ID\_in\_use}_{REUSE}$, there exists a pair $(tn_1, tn_2) \in \mathsf{reuse\_tn\_mapping}$ such that $tn = tn_1$.*
2. *For all $ts \in r.\mathsf{picked\_ts\_ID\_in\_use}_{REUSE}$, there exists a pair $(ts_1, ts_2) \in \mathsf{reuse\_ts\_mapping}$ such that $ts = ts_1$.*
3. *For all $\kappa \in r.\mathsf{picked\_interact\_inst\_in\_use}_{REUSE}$, there exists a pair $(\kappa_1, \kappa_2) \in \mathsf{reuse\_intr\_inst\_mapping}$ such that $\kappa = \kappa_1$.*

Proof. We can easily prove this lemma by induction over the length of executions of Fin. The key idea is that the mapping is overwritten by adding a new mapping relation every time when a new ID (or interaction instance) is picked by a process. □

Now we are ready to prove a simulation relation from Inf to Fin. For this error-free simulation relation, we use TCE-free for Inf, and TCE-TNU-TSU-IIU-free for Fin. We call this simulation relation a TCE/TCE-TNU-TSU-IIU-free simulation relation.

Theorem 10. *The relation $\sim_*$ defined earlier in this section is a TCE/TCE-TNU-TSU-IIU-free simulation relation from Inf to Fin.*

Proof. Let r be a state of Inf and q be a state of Fin such that $r \sim_* q$.

The initial condition and the error-free implication is easy.

We prove the step condition of the simulation relation in the following. Note that we do not have trajectories for Fin and Inf, but only have discrete transitions.

Suppose an action a of Inf is enabled in r. As we described earlier, the keys to the proof are the differences of picked ID lists, expired ID lists, and symbolic timeout times when the time-stamp-estimation trick is used.

1. a uses expiration checks in its effects:
   We need to examine that an expiration check in Inf using $\mathsf{expired\_tn\_ID\_list}_{REUSE}$ conservatively emulates the corresponding expiration check in $Orig$ using $\mathsf{expired\_tn\_ID\_list}$.

   It is easy to check that non-deterministic choice of the expiration check vector is less constrained when we use $\mathsf{expired\_tn\_ID\_list}_1$ and $\mathsf{expired\_ts\_ID\_list}_1$ than when we use $\mathsf{expired\_tn\_ID\_list}_2$ and $\mathsf{expired\_ts\_ID\_list}_2$, if $\mathsf{expired\_tn\_ID\_list}_1 \subseteq \mathsf{expired\_tn\_ID\_list}_2$ and $\mathsf{expired\_ts\_ID\_list}_1 \subseteq \mathsf{expired\_ts\_ID\_list}_2$. We have the following condition because $r \sim_* q$
   $r.\mathsf{expired\_tn\_ID\_list} \supseteq map\_with\_default(\mathsf{reuse\_tn\_mapping}, q.\mathsf{expired\_tn\_ID\_list}_{REUSE}) \wedge$
   $r.\mathsf{expired\_ts\_ID\_list} \supseteq map\_with\_default(\mathsf{reuse\_ts\_mapping}, q.\mathsf{expired\_ts\_ID\_list}_{REUSE})$.
   Therefore, we can always find an action b that has the exactly same expiration vector as a. Since this b has the same results of expiration checks as in a, the effects of b must be the same as a when IDs are mapped, except for the following three aspects:

   (a) [Time-stamp-estimation trick]: When a time-stamp-estimation trick is used, Fin may possibly use a subset of the time stamps set that Inf use for the same timeout setting. This does not heart the simulation relation condition, considering that in Condition

3, we do not need the exact matching, but need just a partial order for symbolic timeout times with a slack added.

(b) [Picked ID sets]: When a timed or timeout variable is assigned a new value in Fin, a time data ID used in the variable before the transition becomes no longer used, and thus is removed from the picked ID sets. This does not heart the simulation relation condition, since in Condition 6, we does not need the exact matching of the picked ID sets, but need just a subset relation.

(c) [Expired ID lists]: When a new time nonce ID or a new time stamp ID is picked in Fin, the ID is removed from the expired ID lists. This does not heart the simulation relation condition, since in Condition 7, we does not need the exact matching of the expired ID sets, but need just a subset relation.

2. a uses expiration checks in the precondition and effects:
The constraint for expiration checks in Fin is less constraining than that in Inf, as we have described above. Therefore, by using the same expiration check vector as a, we can find the corresponding enabled action b. For the effects of b, we can use the same argument as for the first case described above.

3. $a = timeout_i^k$:
We first assert that the corresponding $timeout_i^k$ in Fin is also enabled. It is easy to observe that if a timeout with a slack, and set at symbolic timeout time $\widehat{\tau}$ is not disabled by the scheduler, then a timeout with a slack, and set at $\widehat{\tau_2}$ such that $\widehat{\tau} \succeq \widehat{\tau_2}$, is not disabled by the scheduler as well. We have $r \sim_* q$ and therefore from Condition 3 of $\sim_*$, $timeout_i^k$ is not disabled by the scheduler when it is not disabled in Inf. Considering the above described fact and Condition 4 of $\sim_*$, $b = timeout_i^k$ is enabled in Fin. For the effects of b, we can use the same argument as for the first case described above.

4. The proof for other transitions can be done using the same argument as for the first case described above.

□

The following corollary follows from Theorem 10 using an argument similar to Corollary 8

Corollary 11. *Suppose all executions of Fin that have less than k-many discrete transitions are TCE-TNU-TSU-IIU-free. We call this condition the depth-k global TCE-TNU-TSU-IIU-free condition in Fin. Under this condition, all executions of Inf that have less than k-many discrete transitions are TCE-free, and for any execution β of Inf that has less than k-many discrete transitions, there exists a TCE-TNU-TSU-IIU-free execution γ of Fin such that for any $k \geq 0$, $\beta|_k \sim_* \gamma|_k$, where $\beta|_k$ is the state just after the k-th appearing discrete action in β (when $k = 0$, the initial state), $\gamma|_k$ is the state just after the k-th appearing discrete action in γ (when $k = 0$, the initial state), and $\sim_*$ is the relation defined earlier in this section.*

5.3 Soundness Theorem

From Corollary 8 and Corollary 11, we have the following soundness theorem for TO-abstraction.

Theorem 12. *Suppose all executions of Fin that have less than k-many discrete transitions are TCE-TNU-TSU-IIU-free. We call this condition the depth-k global TCE-TNU-TSU-IIU-free condition in Fin. Under this condition, all executions of $Orig^{Inf}$ that have less than k-many discrete transitions are also TCE-free, and for any execution α of $Orig^{Inf}$ that has less than k-many discrete transitions, there exists a TCE-TNU-TSU-IIU-free execution γ of Fin such that for any $k \geq 0$, $\alpha|_k \sim\sim_* \gamma|_k$, where $\alpha|_k$ is the state just after the k-th appearing discrete action in α (when $k = 0$, the initial state), $\gamma|_k$ is the state just after the k-th appearing discrete action in γ (when $k = 0$, the initial state), and $\sim\sim_*$ is the relation defined by the following: For a state s of $Orig^{Inf}$ and a state q of Fin, $s \sim\sim_* q$ if there exists a state r of Inf such that $s \sim r$ and $r \sim_* q$.*

In the above Theorem 12, $s \sim\sim_* q$ implies that the value of every untimed variable in s is equal to the value of the same untimed variable in q.

6 Case Studies

In this section, we present two case studies of TO-abstraction. The first one is an implementation of the resource-sharing protocol described as Protocol 4. 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 [14]. 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$^{\text{TM}}$ 2 Quad at 2.66 GHz and 4GB memory.

6.1 Case Study: Resource-Sharing Protocol

We applied TO-abstraction to our implementation of Protocol 5. Recall that formal Tempo code of the protocol appears in Figures 1 and 2. The code after time data abstraction appears in Figures 3 and 4.

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).

6.2 Case Study: DHCP Failover Protocol

The *DHCP Failover protocol (DHCP-F)* 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. Upon a request from a client, the DHCP server automatically assign an IP address to the client. The server give an assignment in the form of a "lease": an IP address assignment from the server is associated with its expiration time until which a client is allowed to use the address.

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. Most standard database consistency techniques cannot be used for DHCP-F because they are too slow for this application. For this reason, DHCP-F uses the combination of the two stage assignment scheme. The first assignment is a short assignment that uses a time stamp plus the constant waiting time u. This assignment is fast, requiring no acknowledgment communication, but limits how long the address can be used by one client. The second assignment is slower, requiring explicit acknowledgment from the backup servers, but a client can use one address indefinitely by *renewing* the lease.

DHCP-F is described in an Internet Draft that is over 130 pages long. Part of the length of it is due to the need to deal with many types of concurrent server failures.

In [5], Fan et al. analyzed DHCP-F using the TIOA framework and hand-written proofs of correctness. They *decomposed* the DHCP-F protocol into two sub-protocols: the *leader elector* and the *address assigner*. The leader elector elects one of the backup servers as the next main server when the current main server encounters a failure. The address assigner assigns an IP address to a client upon the request from it. We will explain how this assignment process works in Section 6.2.1.

The leader elector uses a *failure detector*. When the main server encounters a failure, the failure detector notifies the leader electors (implemented in a distributed fashion in each backup server) of this information. The safety property of the leader elector (only one server is elected as the main server at any time) is guaranteed by waiting long enough time since the elector (implemented in the backup server) recovers from a failure. The leader elector sub-protocol and its analysis of the correctness is relatively simple compared to the address assigner, the main task of the DHCP-F protocol. Therefore, in this case study, we will focus mainly on the address assigner protocol of DHCP-F, and we model the leader elector as a simple helper module that notifies the new main server of the fact that the server is elected.

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. In [5], the authors also verified interesting timeliness properties of the protocol as well as the no-duplicated-address-assignment property. Verification of timeliness properties using a machine automated method is a challenging future study.

Because we have to use a model-checker to verify the untimed abstraction, we need to fix the configuration of the system. We look at the minimum interesting configuration, which consists of one main server, one backup server, and two clients (for a possible duplicated assignment).

An actual implementation of the DHCP-F server handle multiple IP addresses concurrently. In this case study, the model handles just one IP address. This is because this concurrent treatment of the IP addresses in one server can be viewed as running multiple threads (or processes) that treat one IP address, and these threads does not affect each other. Therefore, to verify the no-duplicated-address-assignment property of DHCP-F, it is sufficient to focus on one thread that handle one IP address.

6.2.1 TIOA code of DHCP-F

TIOA code for the DHCP-F server is presented in Figures 5 and 6, and TIOA code for the DHCP-F client is presented in Figure 7.

The fail input comes in from the environment at any time. The "leader-elector" helper module outputs the lead action to an alive server with the minimum ID when the module observes a failure in the current execution. In the code in Figure 6, we assume that when a sever encounters a failure, all variables *except for* potlease are reset.

In this code, $\mathsf{lease\_expired}_i$ is the only timeout action in the server and the client. acklease, potlease, and bcast_value are the tree timed variables x_i^1, x_i^2, and x_i^3. timeout represents the only timeout variable t_i^1.

The protocol works as follows in the nominal operation. First, a client broadcasts a request

Vocabulary DHCPF_common_vocab
    % Type declarations
    S2U_Message = enumeration of 'Ack'
    U2S_Message = enumeration of 'Request', 'Renew'
    S2S_Message = enumeration of 'PotleaseWrite', 'PotleaseAck'
    User_ID = Natural$\cup\{\perp\}$
    Server_ID = Natural$\cup\{\perp\}$
    Server_IDs = set of Server_ID Interact_Inst = Natural

Vocabulary DHCPF_Server_vocab
    ServerProgramCounter =
        enumeration of waiting, ack_to_user, send_potlease_ack, bcase_potlease, down, preparing

Automaton DHCPF_Lease_Server$_j$(ε, u: NonNegReal, initialLead: Boolean)
    imports DHCPF_common_vocab, DHCPF_Server_vocab
    signature
        input receive$_j$(i: User_ID, msg: U2S_Message, request_time: NonNegReal, κ: InteractInst)
        input receive$_j$(i: User_ID, msg: U2S_Message, request_time: NonNegReal, κ: InteractInst)
        input receive$_j$(j: Server_ID, msg: S2S_Message, potlease_time: NonNegReal, κ: InteractInst)
        output send$_j$(i: User_ID, msg: S2U_Message, offered_time: NonNegReal, κ: InteractInst)
        output send$_j$(j': Server_ID, msg: S2S_Message, potleaseAckTag: NonNegReal, κ: InteractInst)
        output bcast$_j$(J: Server_IDs, msg: S2S_Message, potlease_time: NonNegReal, κ: InteractInst)
        output lease_expired$_j$ % timeout action
        input fail$_j$ % input from the environment
        input recover$_j$ % input from the environment
        input lead$_j$ % input from a helper module (failure detector)
    states
        clock$_j$: NonNegReal := 0
        timeout_time: NonNegReal := 0
        timeout_is_set: Boolean := false
        pc : LeaseProgramCounter := waiting
        leading: Boolean := initialLead
        current_user: User_ID := $\perp$ %% no current user
        current_interact_inst: InteractInst := $\perp$ %% no current interaction instance
        acklease: NonNegReal := 0
        potlease: NonNegReal := 0
        bcast_value: NonNegReal := 0
        potlease_ack_rcvd: Array[Server_ID, Boolean]
            initially $\forall$ j: Server_ID (potlease_ack_rcvd[i] = false)

Figure 5: TIOA model of a DHCP Failover server, part 1 of 2: Signatures and States

transitions

input receive$_j$(i, 'Request', τ, κ)
  eff if leading$\wedge$current_user $= \perp$
    $\wedge$ pc $=$ waiting then
    current_user $:= i$;
    current_interact_inst $:= \kappa$;
    acklease $:=$ clock$_j + u$;
    potlease $:=$ acklease;
    bcast_value $:=$ max(τ, acklease);
    timeout_time $:=$ potlease $+ \varepsilon$;
    timeout_is_set $:=$ true;
    pc $:=$ ack_to_user; fi

output send$_j$(i, 'Ack', χ, κ)
  pre $\chi =$ acklease $\wedge$
    pc $=$ ack_to_user $\wedge$
    $i =$ current_user $\wedge$
    $\kappa =$ current_interact_inst
  eff pc $:=$ bcast_potlease;

output bcast$_j$(AllServers, 'PotleaseWrite', ω, κ)
  pre pc $=$ bcast_potlease $\wedge$
    $\omega =$ bcast_value $\wedge$
    $\kappa =$ current_interact_inst
  eff pc: $=$ waiting;
    for j':Server_ID:
      potlease_ack_rcvd[j'] := (j' $=$ j);

input receive$_j$(j', 'PotleaseWrite', ω, κ)
  eff if $\neg$ leading $\wedge$ pc $\neq$ down then
    bcast_value $:= \omega$
    potlease $:=$ max(potlease, ω)
    potlease_ack_leader $:= j'$
    pc $:=$ send_potlease_ack fi

output send$_j$(j', 'PotleaseWriteAck', ω, κ)
  pre $\omega =$ bcast_value $\wedge$
    $\kappa =$ current_interact_inst $\wedge$
    pc $=$ send_potlease_ack
  eff pc $:=$ waiting;

input receive$_j$(j', 'PotleaseWriteAck', ω, κ)
  eff if κ=current_interaction_inst$\wedge$
    leading $\wedge$ pc $\neq$ down then
    potlease_ack_rcvd[j'] := true
    if $\forall$ (j:Server_ID): (potlease_ack_rcvd[i]) then
      acklease := max(acklease, ω) fi fi

input receive$_j$(i, 'Renew', τ, κ)
  eff if leading $\wedge$ current_user $= i$
    $\wedge$ pc $\neq$ down then
    current_interact_inst $:= \kappa$;
    acklease $:=$ max(acklease, clock$_j + u$);
    potlease $:=$
      max(potlease, acklease);
    bcast_value $:=$ max(τ, acklease);
    timeout_time $:=$ potlease $+ \varepsilon$;
    timeout_is_set $:=$ true;
    pc $:=$ ack_to_user; fi

output lease_expired$_j$
  pre timeout_time $=$ clock$_j \wedge$
      timeout_is_set
  eff timeout_time $:= 0$;
    timeout_is_set $:=$ false;
    current_user $:= \perp$;
    pc $:=$ waiting;

input fail$_j$
  eff pc $:=$ down
    timeout_time $:= 0$
    timeout_is_set $:=$ false
    leading $:=$ false
    current_user $:= \perp$
    current_interact_inst $:= \perp$
    acklease $:= 0$
    bcast_value $:= 0$
    for j':Server_ID:
      potlease_ack_rcvd[j'] := false

input recover$_j$
  eff pc $:=$ waiting;

input lead$_j$
  eff leading $:=$ true;
    timeout_time $:=$
      max(clock$_j + u + 2\varepsilon$, potlease $+ \varepsilon$);
    timeout_is_set $:=$ true;
    pc $:=$ preparing;

trajectories
  trajdef trajectory
    evolve d(clock$_j$)> 0
    stop when clock$_j \geq$ timeout_time $\wedge$ timeout_is_set

Figure 6: TIOA model of a DHCP Failover server, part 2 of 2: Transitions and Trajectory

```
Vocabulary DHCPF_Client_vocab
    ClientProgramCounter = enumeration of requesting, leasing

Automaton DHCPF_Lease_Client_i(ε, u: NonNegReal)
    imports DHCPF_common_vocab, DHCPF_Client_vocab
    signature
        output bcast_i(J: Server_IDs, msg: U2S_Message, request_time: NonNegReal, κ: InteractInst)
        output send_i(j: Server_ID, msg: U2S_Message, request_time: NonNegReal, κ: InteractInst)
        input receive_i(j: Server_ID, msg: S2U_Message, offered_time: NonNegReal, κ: InteractInst)
        output lease_expired_j
    states
        clock_i: NonNegReal
        timeout: NonNegReal := 0;
        timeout_is_set: Boolean := false;
        pc: ClientProgramCounter := requesting;
        current_server: SERVER_ID := ⊥   %% no current server
        current_interact_inst: Interact_Inst := ⊥   %% no current server

    transitions
        output bcast_i(AllServers, 'Request', τ)
            pre   τ = new_time_nonce() ∧
                  κ = new_interact_inst() ∧
                  pc = requesting
        input receive_i(j, 'Ack', χ)
            eff   if χ > clock_i ∧ κ = current_interact_inst then
                      timeout := χ;
                      timeout_is_set := true;
                      pc := leasing;    fi
        output send_i(j, 'Renew', τ)
            pre   τ = new_time_nonce() ∧
                  j = current_server ∧
                  κ = new_interact_inst() ∧
                  pc = leasing
        output lease_expired_j
            pre   timeout = clock_i ∧ timeout_is_set
            eff   timeout := 0;
                  timeout_is_set := false;
                  pc := idle;
    trajectories
        trajdef trajectory
            evolve    d(clock_i)> 0
            stop when   clock_i = timeout ∧ timeout_is_set
```

Figure 7: TIOA model of a DHCP Failover user

with some time nonce τ, until which a client wants an IP address. It picks a new interaction instance κ for this session, and sends τ with a message type 'Request' and κ. When the main server receives the request, it sends back an acknowledgment with the lease offer with a *short lease*, using $clock_i + u$. It also updates its current interaction instance to κ. Then the server broadcasts the potential lease time (a 'max' of τ and $clock_i + u$) with κ to all backup servers. When a client receives a lease offer, and if the lease time have not expired yet, it accepts the offer and use the IP address until the specified lease time. When a backup server receives the potential lease time from the main server, it updates its own potential lease time, and sends back an acknowledgment to the main server with κ. The main server collects acknowledgements by accepting only acknowledgements for the current session (judged by the interaction instance of the acknowledgements). When the main server has collected acknowledgments from all backup servers, it updates the value of acklease, the variable used to respond a lease-renew request from a client. This implies that the time nonce τ first sent by a client to the main server is used *only after* the main server collects acknowledgement from the backup servers for information of $\max(\tau, clock_i + u)$. When a main server receives a renew request with another time nonce τ_2 from the current client after this "collection" period of the main server, it offers the lease until $\max(\tau, TS_1 + u, clock_i + u)$, where TS_1 is the time stamp taken when the main server first respond to the current client. Then the main server enters the "collection" period for the new time nonce τ_2 ('max'ed with previously accumulated time data in potlease). Similarly, for each following renew message, the client is offered the time nonce for the one-session-previous renew (or request) message, maxed with a new $clock_i + u$ of the main server and accumulated other time data in acklease.

When the main server encounters a failure, the leader-elector detects it, and assigns another server. When a backup server inputs lead, it sets its timeout to 'max' of the potential lease time and '$clock_i + u + 2\varepsilon$' (the time-stamp-estimation trick), so that it times out after all time nonces stored in the potential lease time potlease and all existing time stamps that a client could be using has expired. This intricate use of the time-stamp-estimation trick enables the main server to offer a short lease of $clock_i + u$ *without collecting acknowledgements from backup servers.* Recall that a lease time (time nonce) requested by the client is always used only after collecting acknowledgements from backup servers.

As described above, usage of maximum operations and the time-stamp-estimation trick used in DHCP-F is very subtle, and therefore it is hard to manually analyze whether or not the timeout is set as intended. Thus, we benefit from the power of automated analysis given by TO-abstraction.

6.2.2 Model-Checking DHCP-F Using TO-Abstraction

In this section, we report the automated formal verification of DHCP-F using TO-Abstraction. We also experiment with "mutated" version of the original code to examine the efficiency of bug-finding using TO-Abstraction.

[*Untimed Abstraction of DHCP-F*]

We present the untimed abstraction of DHCP-F. We show in Figures 8, 9, and 10 the untimed version of code described in Figures 5, 6, and 7, respectively. The yellow-colored parts of the code are the parts changed by the abstraction.

Vocabulary Untimed_Abstraction_vocab

% Types for symbolic representation used in an untimed abstraction

SymbolicTimeData = (set of Natural) × (set of Natural)

SymbolicTimeout = (set of Natural) × (set of Natural) × Boolean

Automaton UNTIMED_DHCPF_Lease_Server$_j$

imports DHCPF_common_vocab, DHCPF_Server_vocab, Untimed_Abstraction_vocab

signature

input receive$_j$(i: User_ID, msg: U2S_Message, request_time: SymbolicTimeData, κ: InteractInst)

input receive$_j$(i: User_ID, msg: U2S_Message, request_time: SymbolicTimeData, κ: InteractInst)

input receive$_j$(j: Server_ID, msg: S2S_Message, potlease_time: SymbolicTimeData, κ: InteractInst)

output send$_j$(i: User_ID, msg: S2U_Message, offered_time: SymbolicTimeData, κ: InteractInst)

output send$_j$(j': Server_ID, msg: S2S_Message, potleaseAckTag: SymbolicTimeData, κ: InteractInst)

output bcast$_j$(J: Server_IDs, msg: S2S_Message, potlease_time: SymbolicTimeData, κ: InteractInst)

output lease_expired$_j$ % timeout action

input fail$_j$ % input from the environment

input recover$_j$ % input from the environment

input lead$_j$ % input from a helper module (failure detector)

states

timeout_time: SymbolicTimeout := ({}, {}, false);

timeout_is_set: Boolean := false

pc : LeaseProgramCounter := waiting

leading: Boolean := initialLead

current_user: User_ID := ⊥ %% no current user

current_interact_inst: InteractInst := ⊥ %% no current interaction instance

acklease: SymbolicTimeData := ({}, {});

potlease: SymbolicTimeData := ({}, {});

bcast_value: SymbolicTimeData := ({}, {});

potlease_ack_rcvd: Array[Server_ID, Boolean]

initially ∀ j: Server_ID (potlease_ack_rcvd[i] = false)

global picked_ts_ID_set: set of Natural := {};

global picked_tn_ID_set: set of Natural := {};

global expired_ts_ID_list: set of Natural := {};

global expired_tn_ID_list: set of Natural := {};

Figure 8: Untimed abstraction of a DHCP Failover server, part 1 of 2

transitions

input receive$_j$(i, 'Request', τ, κ)
  eff if leading$\wedge$current_user $= \perp$
      $\wedge$ pc $=$ waiting then
    current_user $:= i$;
    current_interact_inst $:= \kappa$;
    acklease := new_ts_ID(picked_ts_ID_set);
    potlease := acklease;
    bcast_value := symb_max(τ, acklease);
    timeout_time := set_timeout(potlease, true);
    timeout_is_set := true;
    pc := ack_to_user; fi;
    picked_ts_ID_set :=
    one_ts_ID_picked(picked_ts_ID_set)

output send$_j$(i, 'Ack', χ, κ)
  pre $\chi =$ acklease $\wedge$
    pc $=$ ack_to_user $\wedge$
    $i =$ current_user $\wedge$
    $\kappa =$ current_interact_inst
  eff pc := bcast_potlease;

output bcast$_j$(AllServers, 'PotleaseWrite', ω, κ)
  pre pc $=$ bcast_potlease $\wedge$
      $\omega =$ bcast_value $\wedge$
      $\kappa =$ current_interact_inst
  eff pc: $=$ waiting;
    for j':Server_ID:
      potlease_ack_rcvd[j'] := (j' $=$ j);

input receive$_j$(j', 'PotleaseWrite', ω, κ)
  eff if $\neg$ leading $\wedge$ pc $\neq$ down
    then
    bcast_value := ω
    potlease := symb_max(potlease, ω)
    potlease_ack_leader := j'
    pc := send_potlease_ack fi

output send$_j$(j', 'PotleaseWriteAck', ω, κ)
  pre $\omega =$ bcast_value $\wedge$
    $\kappa =$ current_interact_inst $\wedge$
    pc $=$ send_potlease_ack
  eff pc := waiting;

input receive$_j$(j', 'PotleaseWriteAck', ω, κ)
  eff if κ=current_interaction_inst$\wedge$
      leading $\wedge$ pc $\neq$ down then
    potlease_ack_rcvd[j'] := true
    if $\forall$ (j:Server_ID): (potlease_ack_rcvd[i]) then
      acklease := symb_max(acklease, ω) fi fi

input receive$_j$(i, 'Renew', τ, κ)
  eff if leading $\wedge$ current_user $= i$
    $\wedge$ pc $\neq$ down then
    current_interact_inst := κ;
    acklease := symb_max(acklease,
      new_time_stamp_ID(picked_ts_ID_set));
    potlease := symb_max(potlease, acklease);
    bcast_value := symb_max(τ, acklease);
    timeout_time := potlease + ε;
    timeout_is_set := true;
    pc := ack_to_user;
    one_ts_ID_picked(picked_ts_ID_set); fi

output lease_expired$_j$
  pre timeout_is_set
  eff if slack_added(timeout_time)
    then
      expired_ts_ID_list :=
        expired_ts_ID_list $\cup$
        ts_set(timeout_time);
      expired_tn_ID_list :=
        expired_tn_ID_list $\cup$
        tn_set(timeout_time); fi
    timeout_time := ({},{},false);
    timeout_is_set := false;
    current_user := $\perp$;
    pc := waiting;

input fail$_j$
  eff pc := down
    timeout_time := ({},{},false);
    timeout_is_set := false
    leading := false
    current_user := $\perp$
    current_interact_inst := $\perp$
    acklease := ({},{})
    bcast_value := ({},{})
    for j':Server_ID:
      potlease_ack_rcvd[j'] := false

input recover$_j$
  eff pc := waiting;

input lead$_j$
  eff leading := true;
    timeout_time :=
      set_timeout(symb_max(
        ({picked_ts_ID_set},{}),
        potlease),
        true);
    timeout_is_set := true;
    pc := preparing;

Figure 9: Untimed abstraction of a DHCP Failover server, part 2 of 2

Automaton UNTIMED_DHCPF_Lease_Client$_i$

imports DHCPF_common_vocab, DHCPF_Client_vocab, Untimed_Abstraction_vocab

signature

output bcast$_i$(J: Server_IDs, msg: U2S_Message, request_time: SymbolicTimeData, κ: InteractInst)

output send$_i$(j: Server_ID, msg: U2S_Message, request_time: SymbolicTimeData, κ: InteractInst)

input receive$_i$(j: Server_ID, msg: S2U_Message, offered_time: SymbolicTimeData, κ: InteractInst)

output lease_expired$_j$

states

timeout: SymbolicTimeout := ({}, {}, false);

timeout_is_set: Boolean := false;

pc: ClientProgramCounter := requesting;

rcvd_time_expired: Boolean;

current_server: SERVER_ID := $\perp$ %% no current server

current_interact_inst: Interact_Inst := $\perp$ %% no current server

global picked_ts_ID_set: set of Natural := {};

global picked_tn_ID_set: set of Natural := {};

global expired_ts: set of Natural := {};

global expired_tn: set of Natural := {};

output bcast$_i$(AllServers, 'Request', τ, κ)

pre τ = new_time_nonce_ID(picked_tn_ID_set) $\wedge$
κ = new_interact_inst() $\wedge$
pc = requesting

eff picked_ts_ID_set :=
one_ts_ID_picked(picked_ts_ID_set)

input receive$_i$(j, 'Ack', χ, κ)

eff rcvd_time_expired :=
if expired(χ,
expired_ts_ID_list,
expired_tn_ID_list)
then true
else choose(true, false)
if $\neg$ rcvd_time_expired $\wedge$
κ = current_interact_inst
then
timeout := set_timeout(χ, false);
timeout_is_set := true;
pc := leasing; fi

output send$_i$(j, 'Renew', τ, κ)

pre τ = new_time_nonce_ID(picked_tn_ID_set) $\wedge$
j = current_server $\wedge$
κ = new_interact_inst() $\wedge$
pc = leasing

eff picked_ts_ID_set :=
one_ts_ID_picked(picked_ts_ID_set)

output lease_expired$_j$

pre timeout_is_set

eff if slack_added(timeout_time)
then
expired_ts_ID_list :=
expired_ts_ID_list $\cup$
ts_set(timeout_time);
expired_tn_ID_list :=
expired_tn_ID_list $\cup$
tn_set(timeout_time); fi
timeout_time := ({},{},false);
timeout_is_set := false;
pc := idle;

Figure 10: Untimed abstraction of a DHCP Failover client

As we described in 4, all timed and timeout variables now use symbolic representations with time stamp and time nonce IDs. The time-stamp-estimation trick is approximated using picked_ts_ID_set, the time stamp IDs picked thus far in the current protocol execution. The expired time-data list consists of two sublists, expired_ts_ID_list and expired_tn_ID_list, and updated when timeout with a slack ε is performed. Boolean variable rcvd_time_expired is used to determine the truth-value of the expiration check of the form of $clock_i < \chi$, using the expired time-data list.

[*Verification Results*]

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 under the failure assumption that when a server encounters a failure all variables *except for potlease* are reset. The verification time was 350743.7 seconds (97.4 hours).

First Mutation (Different Failure Assumption): Next, we experiment with another failure assumption that when a server encounters a failure, all variables are reset. Under this assumption, we have found a counterexample at depth 17.[13] This counterexample is illustrated in Fig. 11.[14] This scenario is actually realizable in the original timed system as well, as described in the following:

1. Client-1's request contains a large time nonce τ.
2. Write, Write-Ack; and thus the ack-lease is updated (now the maximum contains τ).
3. Client-1 sends a Renew message, and Server 1 receives it. acklease is updated to max(acklease, ts + u) – no smaller than τ. And then, potlease is updated to max(potlease, acklease).
4. A lease offer until max(acklease, ts + u) – no smaller than τ – is sent to Client 1. Client 1 receives it, and update its lease timeout until max(acklease, ts + u).
5. Server 1 encounters a failure, and all variables (especially potlease) is reset. When Server 1 recovers and gets a lead input, it sets the lease timeout to $max(clock_i + u + 2\varepsilon, \mathsf{potlease} + \varepsilon)$, where the value of potlease is 0. Therefore, if τ is larger than the value of $clock_i + u + 2\varepsilon$ at this moment, Server-1 expires before Client-1 expires.
6. Since Server-1 expires, it accepts a request from another client, Client 2, and hence ends up with a duplicated lease.

This counterexample is expected to appear under the assumption of using volatile memory for potlease. This is because potlease stores information of the accumulated information of the potential lease time for the current client. If this information is reset, there is no way the server can retrieve what time nonces the client is now using for its lease time.

[13] This counterexample is found with the configuration that the number of server is one, and the number of clients are two. We are currently running experiment for the case of two servers and two users. We are expecting to find a similar counterexample at depth 20. The illustrated counterexample is for two servers.

[14] We obtained from the counterexample more information than just communication sequence of processes. For example, how potlease and acklease are changed (using symbolic IDs). From these informations, we constructed the realizable scenario in the original real-time model by considering what kind of assumption for the actual values of time nonces are needed to realize the counterexample execution found in the symbolic world.

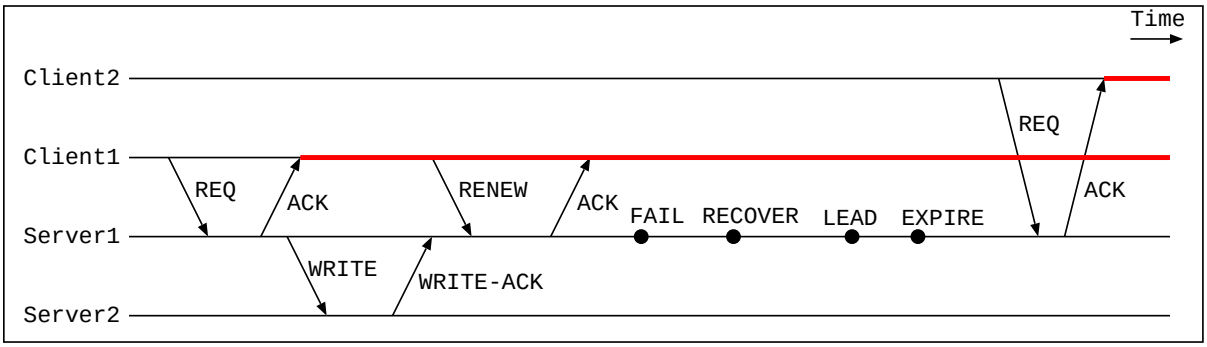

Figure 11: Duplicated lease scenario when all variables are reset by a failure

Second Mutation (Removing Interaction Instance Check): 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 the interaction instance of the received acknowledgement from a backup server to ensure that the received acknowledgement is for the current session. The only change is in the effects of the action receive(j, "WriteAck", ω, κ) of the server automaton. We removed the condition "κ = current_interact_inst" from the if statement. The main server now accepts any "PotleaseWriteAck" acknowledgment as an acknowledgment to the latest "PotleaseWrite" message.

After this mutation, we found a counterexample at depth 17 under the configuration of two servers and two users (the run time was 52851.28 seconds $\sim$ 14.7 hours). This counterexample has a complex scenario, and we believe that it is considerably hard to find by human, or simulations with fixed values of ε and u, and random assignments of time nonces.

The counterexample is depicted in Figure 12. A red line in a client's time line expresses that the client is using a lease.

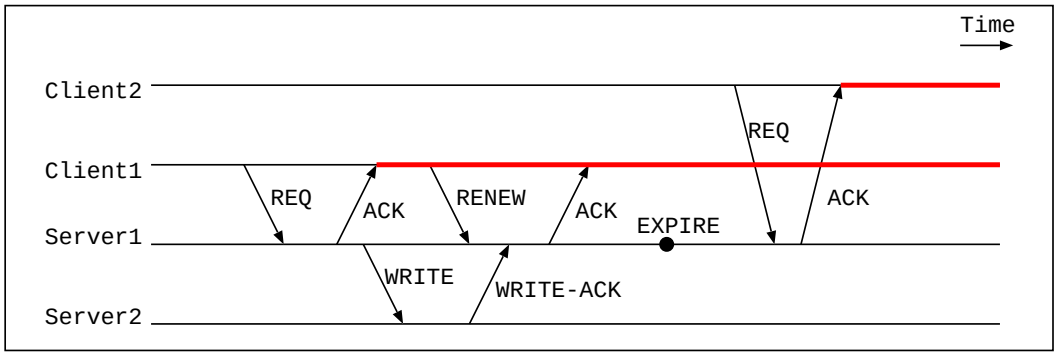

Figure 12: Duplicated lease scenario for the mutated DHCP-F

We now explain why this counterexample is a real counterexample that could occur in the original real-time mutated protocol as well. Suppose the request lease expire time τ from Client-2 is very large and the (potential) lease time covers the entire time frame depicted in Figure 12.

The key of this counterexample is that the main server (Server-1) receives the "WriteAck" message from the backup server (Server-2) between the receipt of the renew message from a client (Client-1) and the acknowledgment for the renew.

We explain the entire scenario in details in the following.

1. Client-1 sends a lease request that contains a large time nonce τ.
2. Server-1 (the main server) receives the request from Client-1, and replies a **short lease** acknowledgment containing the offer until $ts_1 + u$, where ts_1 is a time stamp picked by Server-1 at this point. Note that τ is not included in the short lease. The value of bcast_value changes to $\max(\tau, ts_1 + u)$.
3. Client-1 receives the short lease offer and sends a renew message with a new time nonce τ_2.
4. Server-1 broadcasts the "PotleaseWrite" message with $\max(\tau, ts_1+u)$ (the value of bcast_value) to all backup servers (just one backup in this case).
5. Server-1 receives a renew message from Client-1. Server-1 changes the timeout time (the value of timeout) to $\max(\tau + u, \tau_2 + u) + \varepsilon$, where where ts_2 is a time stamp picked by Server-1 at this point. It also changes the current interaction instance to the one in this renew message.
6. Server-1 then receives "PotleaseWriteAck" from the backup server, Sever-2, *before* acknowledging the renew message from Client-1. The value of current_interact_inst has changed since Server-1 broadcasts the "PotleaseWrite" message. However, the server's program is mutated in such a way that it does not check the interaction instance to decide to accept a received acknowledgment or not. Therefore, Server-1 accepts this "PotleaseWriteAck" from Server-2, and since Server-2 is the only backup server, Server-1 updates the value of acklease used for an acknowledgment to a renew request from the current client. This update is conducted by calculating the maximum of the current value of acklease and the received timed value ω, which contains the large τ.
7. Client-1 receives the renewing-lease offer from Server-1, and set its lease timeout to the maximum that contains τ.
8. Since τ is very large, the timeout set to Server-1 at time $\max(\tau + u, \tau_2 + u) + \varepsilon$ times out before Client-1's lease expires.
9. Client-2 sends a request to Server-1, and Server-1 offers a lease since it has already timed out.
10. Client-2 receives the lease offer and starts using the IP address.

The above described scenario shows that the usage of the maximum operations used in the original DHCP-F protocol are very intricate: the maximum operations used for potlease, acklease, and bcast_value in the main and backup servers control the timing-related behavior of the servers and clients (by offering a lease time), and how they control this timing-related behavior in a correct fashion is very non-trivial. The atomicity assumption for the send and receive actions also affects the correctness of the protocol: The above scenario would not occur if the main server can receive a renew message and send an acknowledgment for it in an atomic

fashion. The atomicity assumption depends on how DHCP-F is implemented. For example, the atomicity granularity we are using in our model is actually realized when broadcasting task and receiving task are operated by two different threads without using lock/unlock. The above discussed facts about this counterexample found for the mutated protocol strongly suggests that we need to use an automated exhaustive analysis technique such as TO-abstraction for this kind of real-time protocols.

7 Conclusion

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] B. W. Lampson, N. A. Lynch, and J. F. Soegaard-Andersen. Correctness of at-most-once message delivery protocols. In *Formal Description Techniques, VI (Proceedings of the IFIP TC6/WG6.1 Sixth International Conference on Formal Description Techniques — FORTE'93, Boston, MA, October 1993)*, pages 385–400, 1994.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] S. Umeno and N. Lynch. Supplimental files (sal code, conterexample traces) for the technical report. The files can be obtained from `http://people.csail.mit.edu/umeno/to_abst_tr/`.

[15] 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.

[16] 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.