A Timeless Model for the Verification of Quasi-Periodic Distributed Systems

Maryam Dabaghchian
School of Computing
University of Utah
Salt Lake City, UT, USA
maryam@cs.utah.edu

Zvonimir Rakamarić
School of Computing
University of Utah
Salt Lake City, UT, USA
zvonimir@cs.utah.edu

ABSTRACT

A cyber-physical system often consists of distributed multi-rate periodic processes that communicate using message passing; each process owns a local clock not synchronized with others. We call such systems quasi-periodic distributed systems. Traditionally, one would model them using timed automata, thereby having to deal with high-complexity verification problems. Recently, several researchers proposed discrete-time abstractions based on the calendar model to make the verification more tractable. However, even the calendar model contains a notion of time in the form of a global clock. We propose a novel, timeless computation model for quasi-periodic distributed systems to facilitate their verification. The main idea behind our model is to judiciously replace synchronization using a global clock and calendar with synchronization over lengths of message buffers. We introduce a simple domain-specific language for programming of such systems and use it to formalize the semantics of both the calendar and timeless model. Then, we prove that our timeless model is an overapproximation of the calendar model. Finally, we evaluate our timeless model using several benchmarks.

CCS CONCEPTS

• **Theory of computation** → **Models of computation**; **Distributed computing models**; • **Computer systems organization** → *Robotics*; *Embedded systems*; *Real-time systems.*

KEYWORDS

quasi-periodic distributed systems, finite-state systems, model checking, timeless model, calendar model

ACM Reference Format:
Maryam Dabaghchian and Zvonimir Rakamarić. 2019. A Timeless Model for the Verification of Quasi-Periodic Distributed Systems. In *17th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '19), October 9–11, 2019, La Jolla, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3359986.3361201

MEMOCODE '19, October 9–11, 2019, La Jolla, CA, USA

ACM ISBN 978-1-4503-6997-8/19/10…$15.00
https://doi.org/10.1145/3359986.3361201

1 INTRODUCTION

A cyber-physical system often consists of multiple processes, where each process is periodically activated at its preset nominal rate. Processes are approximately synchronized using their local clocks with a bounded *drift* from a global reference clock (i.e., each local clock might slightly diverge from the global reference clock). At each activation, a process performs its computation and communicates with other processes using the publish-subscribe messaging paradigm in a non-blocking manner. In the publish-subscribe messaging paradigm, there are a number of *topics* (or message channels) to which a process can publish or subscribe to. Typically, each topic has one publisher (to avoid network jamming) that sends messages to it and several subscribers that receive them. We call such a system a *quasi-periodic distributed system* (QPDS). Robot Operating System [33] is an example of a well-known and widely used framework that employs this paradigm.

Model checking [12, 32] is a class of verification techniques typically employed in the verification of QPDSs. Due to its continuous-time aspects, a QPDS would traditionally be modeled using timed automata [2], and verified using model checkers such as UPPAAL [5]. However, due to the high complexity of the timed automata model checking problem, it is unlikely that this approach will scale to large systems. Hence, researchers have been exploring alternative, more efficient modeling approaches, such as the discrete-time calendar model [18, 19, 29, 35]. The calendar model maintains a set containing future discrete events with the time at which they should be scheduled. When the next schedulable event is removed from the set, the system time is updated as a discrete step. However, though individual steps are discrete, even this model maintains a notion of time in the form of a global real-valued clock, which means that it is still beyond the reach of classical model checkers for finite-state systems. To address this problem, we propose a timeless finite-state model for QPDSs.

We introduce a simple domain-specific language for programming of QPDSs. Our language contains annotations that allow for users to specify key parameters of their system, such as the periods of all processes, drifts, sizes of message buffers, and expected numbers of messages at every process activation. We use our language to formalize the semantics of the calendar model and our timeless finite-state model. The main idea behind our model is to judiciously replace calendar-based synchronization that employs a global clock with synchronization that uses predicates over message buffers. We achieve this by constructing a constraint satisfaction problem using the user-provided parameters of the system, such as periods, drifts, delays, and buffer sizes. If the generated constraints are satisfiable, then the system is amenable to overapproximation (i.e.,

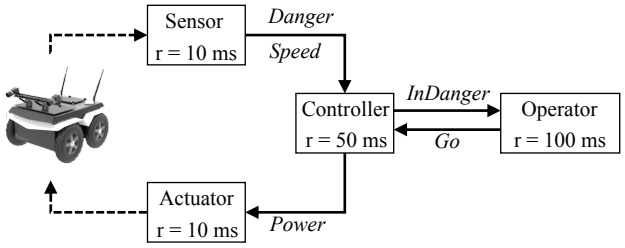

Figure 1: Semi-autonomous ground vehicle architecture.

sound modeling) using our timeless model; otherwise, a user has to adjust the system parameters. Furthermore, we generate buffer predicates from the system parameters, and use them for process synchronization in the timeless model. For example, we ensure that a publish statement cannot proceed if there is a subscriber with a full receiving message buffer, where the message buffer size satisfies our constraint problem. Finally, we prove that our timeless model is an overapproximation (i.e., sound abstraction in terms of safety properties) of the respective calendar model.

We summarize our main contributions as follows:

- We introduce a simple domain-specific programming language for QPDSs, and use it to formalize the semantics of the previously introduced calendar model.
- We are the first to propose and formalize a timeless model for QPDSs that can be verified using classical model checkers for finite-state systems.
- We define a constraint satisfaction problem that checks whether a QPDS is amenable to sound modeling using the user-provided parameters of the system.
- We prove that our timeless model is an overapproximation of the calendar model.
- We develop the timeless model of several benchmarks in SPIN to evaluate our approach.

2 PRELIMINARIES

In this section, we introduce a simple example, our program syntax, and basic definitions of QPDSs.

2.1 Simple Example

Figure 1 shows an example QPDS modeling a semi-autonomous ground vehicle [29]. It consists of four processes called *Sensor*, *Operator*, *Controller*, and *Actuator*, where r denotes their periods (i.e., a process activates every r milliseconds). Arrows between processes capture the publisher-subscriber relationships. For example, *Sensor* publishes to topics *Speed* and *Danger*, while *Controller* reads messages from those two topics and publishes to topics *InDanger* and *Power*. Note that *Controller* can read and perform its computation using multiple messages from the *Speed* or *Danger* topics, as its activation period is five times the *Sensor*'s activation period.

To illustrate a possible behavior of the system, suppose that *Sensor* publishes the current velocity to topic *Speed* and a Boolean value to topic *Danger* notifying about a detected obstacle. By taking into account the nominal periods and drifts of *Sensor* and *Controller*, we can observe that there will be at least four messages present in the *Controller*'s buffers when it activates. Hence, the controller can compute the needed actuator power based on the last four velocity values. If the velocity goes out of the allowed range or if an obstacle is detected, *Controller* publishes value 0 to topic *Power*, thereby stopping the vehicle. As we can observe from this example behavior, a precise analysis of this system is complex since it involves reasoning about the continuous local clocks, periods, and drifts. As we show in this paper, it is possible to construct finite-state models of such systems that get rid of the complex continuous-time aspects, while preserving soundness of the verification.

```
prog ::= delay dmin dmax
           topic+ proc+
topic ::= topic tid
proc ::= process pid annot {stmt}
annot ::= period r drift ρ pubsub+
pubsub ::= publishes tid
         | subscribes tid size new max_lost
stmt ::= read m := tid
       | publish tid m
       | return
       | stmt; stmt
```

Figure 2: Program syntax.

2.2 Program Syntax

Figure 2 shows our program syntax, which enhances a simple While language (an imperative programming language with while loops, assignments, and conditionals [26]; not shown for simplicity) with features to support periodic process definition and publish/subscribe communication. A program consists of multiple processes communicating over topics. Annotation delay *dmin dmax* defines the minimum and maximum bounds of the message transmission delay for all topics. (Note that our approach can be extended to allow for different delay bounds for each topic.) Each topic has only one publisher process, whereas it can have multiple subscribers. We assign a unique identifier *tid* to each topic and *pid* to each process. A process also contains annotations and a top-level statement.

Annotations allow for a user to specify the parameters of its QPDS, based on which we ensure it can be soundly abstracted using our timeless model (see Section 4.1). Annotation period r sets the period of the corresponding process, while drift ρ sets its maximum drift as a fraction of r where $0 \leq \rho < 1$. Annotation publishes *tid* registers it as the only publisher to topic *tid*, while subscribes *tid size new max_lost* registers it as a subscriber to topic *tid* with *size* specifying the receiving buffer size, *new* the minimum number of messages required on the topic for the process to perform its local computation, and *max_lost* the maximum number of consecutive messages that can be lost due to an insufficient buffer size (i.e., when a message buffer gets full we drop the oldest messages in the buffer to accommodate further incoming messages).

```
delay .1 .2
topic Speed, Danger, InDanger, Go, Power
process Sensor
  period 10 drift .1
  publishes Speed
  publishes Danger
{
  //evaluate speed and obstacle
  publish Speed speed;
  publish Danger obstacle;
  return
}
process Controller
  period 50 drift .1
  publishes InDanger
  publishes Power
  subscribes Danger 6 4 1
  subscribes Speed 7 4 0
  subscribes Go 1 0 0
{
  read obstacle := Danger;
  read go := Go;
  read speed := Speed;
  // evaluate power
  publish Power power;
  publish InDanger obstacle;
  return
}
```

Figure 3: Source code excerpt of our simple example.

Statement read $m := tid$ retrieves the first message received on topic *tid*, removes it from the sequence of received messages, and stores it into local message variable *m*. Statement publish *tid m* publishes message *m* on topic *tid*. A process can publish only one message per topic during each activation. Statement return denotes the end of an execution period of a process. Figure 3 shows example source code for the motivating example from Section 2.1.

2.3 Basic Definitions

A process $p \in Procs$ is a tuple $\langle pid, r, \rho, publish, subscribe \rangle$, where *pid* is a unique identifier, r is the period at which the process is activated, ρ is the maximum clock drift, *publish* is the set of topics (defined next) on which p publishes, and *subscribe* is the set of topics to which p subscribes. We refer to each tuple element using [] notation, e.g., $p[r]$ denotes the nominal period of p. A topic $t \in Topics$ is a tuple $\langle tid, pub, subs \rangle$, where *tid* is a unique identifier, *pub* is the publisher process, and *subs* is the set of subscriber processes. We denote with $m \in Msgs$ a message of an unspecified type transmitting between processes in the system. Let $\kappa : (Procs \times Topics) \rightharpoonup Msgs^*$ be a partial mapping capturing the contents of buffers; then, $\kappa(p, t)$ is the sequence of received messages in the buffer corresponding to topic t in process p before it gets activated. Let $len : Msgs^* \rightarrow \mathbb{Z}_{\geq 0}$ be a function that returns the number of messages in a buffer. Then, $len \circ \kappa$ returns the number of messages in a receiving buffer. Let $lost : (Procs \times Topics) \rightharpoonup \mathbb{Z}_{\geq 0}$ be a partial mapping capturing the number of messages lost in transit on a topic due to a full receiving buffer; then, $lost(p, t)$ is the number of lost messages published to process p on topic t. Let $size : (Procs \times Topics) \rightharpoonup \mathbb{N}$ be a partial mapping capturing the sizes of buffers; then, $size(p, t)$ is the size of the buffer corresponding to topic t in process p. Furthermore, let $max\_loss : (Procs \times Topics) \rightharpoonup \mathbb{Z}_{\geq 0}$ be a partial mapping, where $max\_loss(p, t)$ is the maximum number of consecutive messages that can be lost due to an insufficient message buffer size while transmitting from a publisher process to the subscriber process p through topic t.

We denote the set of all statements with *Stmts*, while for every process p we denote its top-level statement with S_p. Then, $S : Procs \rightarrow Stmts$ captures the remaining statement to be executed in a processes. Let *Variables* be the set of all local variables in all processes, and $Variables|_p$ the set of local variables in p. For simplicity, we assume that all variables have the same domain *Domain*. We insist that every process has one local buffer q_{tid} per topic it is subscribed to, which we use to copy locally the contents of message buffers. Then, we define program store $\mu : Variables \rightarrow Domain$ such that $\mu(x)$ is the valuation of variable x. Finally, we define a program state $\sigma \in States$ as a tuple $\langle \mu, \kappa, lost \rangle$.

3 CALENDAR MODEL

In this section, we present the calendar model as a discrete-time quasi-synchronous abstraction of QPDSs, thereby allowing for the verification of real-time systems without relying on continuously varying clocks [18, 19, 29].

3.1 Soundness Properties of QPDSs for Quasi-Synchrony

A QPDS consists of a finite set of processes *Procs*. Each process $p \in Procs$ periodically performs a task with a nominal period $p[r]$ and a possible jitter due to the clock drift bounded by $p[\rho]$. Hence, the actual period at any step varies between $p[r](1 - p[\rho])$ and $p[r](1 + p[\rho])$. We allow for messages to be transmitted between processes with a delay between the minimum delay D_{min} and the maximum delay D_{max}, where $0 \leq D_{min} \leq D_{max}$.

Caspi [9–11] first introduced the quasi-synchronous approach as a discrete-time abstraction of single-rate QPDSs. Then, Larrieu and Shankar [28] presented a computation model for multi-rate QPDSs based on this quasi-synchronous abstraction. A QPDS has to satisfy the following properties for their model to be a sound abstraction of it.

- (P1) A message published by process p' is processed by a subscriber process p within time D_{min} and $D_{max} + p[r](1+p[\rho])$ if it is not lost.
- (P2) Messages are delivered in their publishing order, which the constraint $D_{max} < D_{min} + p[r](1 - p[\rho])$ ensures for every publisher process p.
- (P3) Buffer sizes have to be sufficient to limit the maximum number of lost messages: $size(p, t) \geq \frac{p[r](1+p[\rho])+D_{max}-D_{min}}{p'[r](1-p'[\rho])} - max\_loss(p, t)$. The fraction captures the scenario where a

publisher is activated as many times as possible and a subscriber as few times as possible, so that the subscriber delivers the maximum number of messages (see Figure 6).

In a complementary paper, Baudart et al. [4] prove that the original idea of quasi-synchronous abstraction is not sound, though soundness is recoverable. More recently, Baudart [3] presented soundness conditions for a discrete model of a QPDS, where a discrete model is called N/M-quasi-synchronous if a process is activated at most N times between M consecutive activations of another process. He proves that a QPDS is N/M-quasi-synchronous if it satisfies the following two properties in addition to P1–P3 above.

(P4) A process p can receive at most N messages published by p' between M consecutive activations, which the following constraint ensures: $N(p'[r](1 - p'[\rho])) + D_{min} \geq (M - 1)(p[r](1 + p[\rho])) + D_{max}$. In addition, a process p is activated at most N times between M messages received from p', which the following constraint ensures: $N(p[r](1 - p[\rho])) + D_{min} \geq (M - 1)(p'[r](1 + p'[\rho])) + D_{max}$. Note that, as in previous work [28], we assume that $M = 2$; then, the former constraint in this property reduces to P3, where $N = size(p, t) + max\_loss(p, t)$, and is therefore its generalization.

(P5) The communication topology of the system satisfies the following conditions: (1) every elementary cycle in the undirected variant of the system's communication graph (called u-cycle) is a cycle in the directed graph as well, or the number of its edges in the both directions is the same (called balance u-cycle), or $D_{max} = 0$; (2) there is no balanced u-cycle in the communication graph, or $D_{min} = D_{max}$; (3) there is no cycle in the communication graph, or for every cycle c of length[1] $|c|$, $p[r](1 - p[\rho]) \geq |c|D_{max}$, where p has the minimum period among all processes in the cycle.

If a QPDS satisfies these properties, it is amenable to overapproximation using our calendar model since the model is a quasi-synchronous abstraction. Furthermore, we also use these properties in our timeless model to compute the needed buffer sizes and minimum numbers of messages each process has in its buffers at the time of activation; we leverage these to achieve proper synchronization without relying on time (see Section 4).

3.2 Program Semantics

We denote with $ct \in \mathbb{R}_{\geq 0}$ the current time with respect to an ideal reference clock. An event $e \in Procs \times Topics \times Msgs$ is either a message delivery or a process activation event. A message delivery event $\langle p, t, m\rangle$ denotes the delivery of message m from topic t by process p. A process activation event $\langle p, \epsilon, \epsilon\rangle$ denotes the activation of process p. Then, the calendar C is a set of pairs $\langle e, \tau\rangle$, where for an event e the timeout $\tau \in \mathbb{R}_{\geq 0}$ denotes the earliest future time at which e is scheduled. The calendar maintains scheduled events and preserves the invariant $\forall \langle e, \tau\rangle \in C\,.\,ct \leq \tau$. We use $min(C)$ to denote the earliest timeout in the calendar. The enabled set $Enabled_{DT} = \{p \mid \langle\langle p, \epsilon, \epsilon\rangle, ct\rangle \in C\}$ is the set of all processes that can be activated at the current time.

[1]We define a length of a cycle in a directed graph as usual as the number of edges in the cycle.

A *configuration* is a tuple $c = \langle S, \sigma, C, ct\rangle$, where S is the remaining statements to be executed, σ is a program state, C is the calendar, and ct is the global current time. An *execution* is a sequence of configurations $c_1c_2c_3\ldots$, where $c_1 = \langle S_1, \langle \mu_1, \kappa_1, lost_1\rangle, C_1, 0\rangle$ is the initial configuration and $c_i \xrightarrow[DT]{} c_{i+1}$ as per the calendar-based operational semantics rules (see Figure 4). Initially, S_1 maps every process p to an empty statement ϵ, μ_1 maps every variable to a default value, κ_1 maps every topic buffer to an empty sequence of messages in all subscriber processes, and $lost_1$ maps the number of lost messages for every pair of a subscriber and topic to 0. Furthermore, C_1 contains a pair $\langle\langle p, \epsilon, \epsilon\rangle, p[r]d\rangle$ for every process p, where $d \in [1 - p[\rho], 1 + p[\rho]]$ and hence $p[r]d$ is the initial activation timeout for p.

Figure 4 gives the operational semantics rules of $\xrightarrow[DT]{}$. Rule Delay ensures that if all events in the calendar are scheduled for a future time, then the current time advances to the timeout of the earliest event in the calendar. Rule Deliver captures message delivery, where $q \oplus m$ denotes the buffer obtained by extending buffer q with message m. Rule Deliver-Loss captures message loss, where $q[1{:}]\oplus m$ denotes the buffer obtained by dropping the oldest message from buffer q and extending it with message m. Rule Activate starts an enabled process. It copies the contents of all the receiving message buffers into the corresponding local buffers, denoted by q_t for every subscribing topic t (see Section 2.3); then, it empties the receiving buffers and resets the number of lost messages for every subscribing topic. Rule Read reads a received message from a non-empty local message buffer q_t by removing it from the buffer and storing it into a local message variable m. We use the notation $q[0]$ to refer to the first element in the buffer object, and $q[1{:}]$ to refer to the sequence of elements excluding the first one. Rule Read-Empty takes care of reading from an empty message buffer by storing a special value *null* into local variable m. Rule Publish adds an appropriately delayed message delivery event to the calendar for every subscriber of the topic. Rule Reset ensures the periodicity of a processes by updating its timeout in the calendar based on the specified period and drift.

Since there is infinitely many choices between any two real-valued clock valuations, a real-time system contains infinite-delay transitions in addition to discrete transitions. Hence, researchers have introduced discrete-time models, such as the calendar model, to reduce the infinite-delay transitions between two discrete transitions to finite-delay transitions, and have effectively used them to verify the correctness of real-time systems. However, even finite-delay transitions in the calendar model introduce extraneous states since they only update the global clock. In addition, the monotonicity of the global real-valued time variable (i.e., clock) still ultimately results in an infinite-state system, which hinders verification using classical model checkers. Hence, in the next section, we propose a novel, timeless model that employs predicates over the lengths of buffers, instead of the real-valued global time, for process synchronization. Our model has two key advantages over the calendar model due to the elimination of time: (1) it has no time and delay transitions, and thereby it reduces the program state space; (2) it is a finite-state model because it does not contain a monotonically-increasing global time variable, thereby allowing for

$$\textsc{Delay}\quad \frac{ct < min(C)}{\langle S, \sigma, C, ct\rangle \xrightarrow[DT]{} \langle S, \sigma, C, min(C)\rangle}$$

$$\textsc{Deliver}\quad \frac{\langle\langle p, t, m\rangle, ct\rangle \in C \qquad q = \kappa(p, t) \qquad len \circ q < size(p, t)}{\langle S, \langle \mu, \kappa, lost\rangle, C, ct\rangle \xrightarrow[DT]{} \langle S, \langle \mu, \kappa', lost\rangle, C', ct\rangle} \quad \kappa' = \kappa[(p, t) \mapsto q \oplus m] \quad C' = C \setminus \{\langle\langle p, t, m\rangle, ct\rangle\}$$

$$\textsc{Deliver-Loss}\quad \frac{\langle\langle p, t, m\rangle, ct\rangle \in C \qquad len \circ \kappa(p, t) = size(p, t) \qquad q = \kappa(p, t) \qquad l = lost(p, t)}{\langle S, \langle \mu, \kappa, lost\rangle, C, ct\rangle \xrightarrow[DT]{} \langle S, \langle \mu, \kappa', lost'\rangle, C', ct\rangle} \quad \kappa' = \kappa[(p, t) \mapsto q[1{:}] \oplus m] \quad C' = C \setminus \{\langle\langle p, t, m\rangle, ct\rangle\} \quad lost' = lost[(p, t) \mapsto l + 1]$$

$$\textsc{Activate}\quad \frac{p \in Enabled_{DT}}{\langle S[p \mapsto \varepsilon], \langle \mu, \kappa, lost\rangle, C, ct\rangle \xrightarrow[DT]{} \langle S[p \mapsto S_p\ \langle \mu', \kappa', lost'\rangle, C, ct\rangle} \quad \mu' = \mu[q_t \mapsto \kappa(p, t)] \text{ for every } t \in p[subscribe] \quad \kappa' = \kappa[(p, t) \mapsto [\,]] \text{ for every } t \in p[subscribe] \quad lost' = lost[(p, t) \mapsto 0] \text{ for every } t \in p[subscribe]$$

$$\textsc{Read}\quad \frac{m \in Variables|_p \qquad q = \mu(q_t) \qquad q \neq [\,]}{\langle S[p \mapsto \text{read } m := tid; stmt], \langle \mu, \kappa, lost\rangle, C, ct\rangle \xrightarrow[DT]{} \langle S[p \mapsto stmt], \langle \mu', \kappa, lost\rangle, C, ct\rangle} \quad \mu' = \mu[m \mapsto q[0], q_t \mapsto q[1{:}]]$$

$$\textsc{Read-Empty}\quad \frac{m \in Variables|_p \qquad \mu(q_t) = [\,]}{\langle S[p \mapsto \text{read } m := tid; stmt], \langle \mu, \kappa, lost\rangle, C, ct\rangle \xrightarrow[DT]{} \langle S[p \mapsto stmt], \langle \mu', \kappa, lost\rangle, C, ct\rangle} \quad \mu' = \mu[m \mapsto null]$$

$$\textsc{Publish}\quad \frac{d \in [D_{min}, D_{max}]}{\langle S[p \mapsto \text{publish}(t, m); stmt], \sigma, C, ct\rangle \xrightarrow[DT]{} \langle S[p \mapsto stmt], \sigma, C', ct\rangle} \quad C' = C \cup \{\langle\langle p', t, m\rangle, ct + d\rangle \mid p' \in t[subs]\}$$

$$\textsc{Reset}\quad \frac{d \in [1 - p[\rho], 1 + p[\rho]]}{\langle S[p \mapsto \text{return}], \sigma, C, ct\rangle \xrightarrow[DT]{} \langle S[p \mapsto \varepsilon], \sigma, C', ct\rangle} \quad C' = C \setminus \{\langle\langle p, \epsilon, \epsilon\rangle, ct\rangle\} \cup \{\langle\langle p, \epsilon, \epsilon\rangle, ct + p[r]d\rangle\}$$

Figure 4: Operational semantics of the calendar-based transition relation $\xrightarrow[DT]{}$.

efficient verification using classical model-checkers for finite-state systems.

4 TIMELESS MODEL

In this section, we propose a timeless model for QPDSs. The main idea behind this model is to achieve proper synchronization between processes using predicates over message buffers (e.g., a process is activated only when a certain number of messages is present in its buffer), instead of relying on a clock. First, we formalize a constraint satisfaction problem using the user-provided parameters of the system, such as periods, drifts, delays, and buffer sizes. If the generated constraints are satisfiable, then the system is amenable to sound modeling using our timeless model. Second, we generate buffer predicates from the system parameters, and use them for process synchronization in the operational semantics of the timeless model.

4.1 Constraint Satisfaction Problem

Figure 5 shows our constraint satisfaction problem. Its inputs are all periods, drifts, delays, sizes of buffers, minimum numbers of messages required in buffers at activations, and maximum numbers of messages that could be lost due to insufficient buffer sizes. These are all specified by a user using the annotations our language provides. The satisfiability of the constraints implies that the QPDS it encodes can be overapproximated using our timeless model. Note that while we present the constraints using quantifiers, all are over finite domains and can hence be instantiated ahead of time. Next, we motivate each constraint and related them to the properties described in Section 3.1.

Constraint (1) guarantees that the publishing order of messages is preserved, as specified by property P2. A message delay, which in the worst case is D_{max}, must be less than the time required for the next message delivery, which in the best case is when a publisher process p' is activated at its minimum period and the message is delayed by D_{min}. Constraint (2) guarantees that the sizes of buffers are sufficient to bound the maximum number of lost messages, as specified by property P3. Since in our timeless model we use the sizes of buffers for process synchronization, they have to be equal (and not greater than!) the smallest integer that satisfies property P3 (see Figure 6). Constraint (3) guarantees that the minimum required number of messages is present in the receiving buffers when a process is activated. (We formally introduce $min\_new(p, t)$ in the next section.) Figure 7 illustrates this constraint by showing a scenario where a publisher is activated as few times and a subscriber as many times as possible, which is when the subscriber receives the minimum number of messages before it is activated.

Constraints (4), (5), and (6) correspond to property P5. Let *Cycles* be the set of all subsets of processes that form an elementary cycle in the undirected variant of the communication graph of the system. We assume that the weight of a clockwise edge in a cycle is 1 and otherwise it is -1. Then, function $weight : Cycles \to \mathbb{Z}_{\geq 0}$ maps cycles to their weights, and $weight(c)$ is the absolute value of the sum of weights of its edges. Constraint (4) guarantees that if there is an unbalanced u-cycle (i.e., a cycle whose weight is greater than 0 and less than the number of processes in the cycle) in the communication graph, transmission delay must be zero. Constraint (5) guarantees that if there is a balanced u-cycle (i.e., a cycle whose weight is 0) in the communication graph, D_{min} and D_{max} must be equal. Finally, constraint (6) guarantees that for every cycle in the communication graph, the minimum period of all processes in the cycle must be greater than or equal to $|c|D_{max}$.

4.2 Program Semantics

Let $\zeta : (Procs \times Topics) \rightharpoonup Msgs^*$ be a partial mapping capturing the contents of communication channels (i.e., messages in transit); then, $\zeta(p, t)$ is the sequence of messages published to topic t and waiting in a communication channel to be delivered to process p. Let $min\_new : (Procs \times Topics) \rightharpoonup \mathbb{Z}_{\geq 0}$ be a partial mapping specifying the minimum number of messages that must be present in the subscriber process receiving buffer of a topic when that subscriber is activated. The enabled set $Enabled_{NT} = \{p \mid \forall t \in p[subscribe]\,.\, len \circ \kappa(p, t) \geq min\_new(p, t)\}$ is the set of all processes that can be activated, i.e., all their receiving buffers have the required minimum number of messages. The publishable predicate

$$\forall t \in Topics\,.\, p = t[pub] \wedge D_{max} < p[r](1 - p[\rho]) + D_{min} \quad \text{(P2)} \tag{1}$$

$$\forall t \in Topics, p \in t[subs]\,.\, p' = t[pub] \;\wedge size(p,t) + max\_loss(p,t) = \left\lceil \frac{p[r](1+p[\rho])+D_{max}-D_{min}}{p'[r](1-p'[\rho])} \right\rceil \quad \text{(P3)} \tag{2}$$

$$\forall t \in Topics, p \in t[subs]\,.\, p' = t[pub] \;\wedge min\_new(p,t) = \left\lfloor \frac{p[r](1-p[\rho])-(D_{max}-D_{min})}{p'[r](1+p'[\rho])} \right\rfloor \quad \text{(P3)} \tag{3}$$

$$(\forall c \in Cycles\,.\, weight(c) = 0 \;\vee\; weight(c) = |c|) \;\vee\; D_{max} = 0 \quad \text{(P5)} \tag{4}$$

$$(\forall c \in Cycles\,.\, weight(c) > 0) \;\vee\; D_{min} = D_{max} \quad \text{(P5)} \tag{5}$$

$$(\forall c \in Cycles\,.\, weight(c) < |c|) \;\vee\; (\forall c \in Cycles\,.\, weight(c) = |c| \implies \min\{p[r](1 - p[\rho]) \mid p \in c\} \geq |c| D_{max}) \quad \text{(P5)} \tag{6}$$

Figure 5: Constraint satisfaction problem that ensures a QPDS is amenable to overapproximation using our timeless model. The problem inputs are r, ρ, D_{min}, D_{max}, *size*, *min_new*, *max_loss*, *Topics*, and *Cycles*.

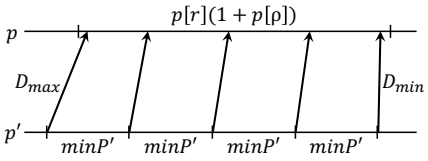

Figure 6: Maximum number of messages p receives from p' between two consecutive activations.

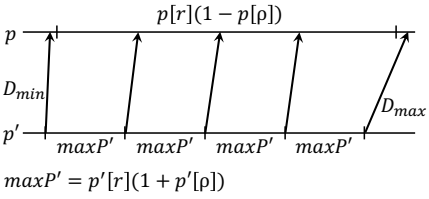

Figure 7: Minimum number of messages p receives from p' between two consecutive activations.

determines if a process can publish to a topic given the current state of its buffers, channels, and the number of lost messages, i.e., whether publishing would violate the maximum number of possible lost messages: $Publishable(t, \kappa, \zeta, lost) \equiv \forall p \in t[subs]\,.\, len \circ \kappa(p,t) + len \circ \zeta(p,t) + lost(p,t) < size(p,t) + max\_loss(p,t)$. A configuration is a tuple $c = \langle S, \sigma, \zeta \rangle$, where S is the remaining statements to be executed, σ is a program state, and ζ is the communication channels. An execution is a sequence of configurations $c_1 c_2 c_3 \ldots$, where $c_1 = \langle S_1, \langle \mu_1, \kappa_1, lost_1 \rangle, \zeta_1 \rangle$ is the initial configuration and $c_i \xrightarrow[NT]{} c_{i+1}$ as per the timeless operational semantics rules (see Figure 8). Initially, S_1 maps every process p to an empty statement ε, μ_1 maps every variable to a default value, κ_1 maps every topic buffer to an empty sequence of messages in all subscriber processes, $lost_1$ maps the number of lost messages for every pair of a subscriber and topic to 0, and ζ_1 maps every pair of a subscriber and topic to an empty sequence of messages.

Figure 8 gives the operational semantics rules of $\xrightarrow[NT]{}$. Apart from Delay that updates the global clock, there is a corresponding transition rule in $\xrightarrow[NT]{}$ for every rule in $\xrightarrow[DT]{}$ (see Figure 4). In the next section, we show that our timeless model is an overapproximation of the calendar model.

5 PROOF OF OVERAPPROXIMATION

In this section, we prove that our timeless model is an overapproximation of the calendar model, which has been used for efficient modeling of QPDSs. We do this by showing that when starting from equivalent initial states, the set of all possible executions in the calendar model is a subset of the set of all possible executions in the timeless model.

Definition 1 (State Equivalence). *Two states $\sigma = \langle \mu, \kappa, lost \rangle$ and $\sigma' = \langle \mu', \kappa', lost' \rangle$ are equivalent, denoted by $\sigma \equiv \sigma'$, if and only if $\mu = \mu'$, $\kappa = \kappa'$, and $lost = lost'$.*

Definition 2 (C-ζ Equivalence). *The calendar C and the communication channels ζ are equivalent, denoted by $C \equiv \zeta$, if and only if:*

1. $\forall t \in Topics, p \in t[subs], i \in \mathbb{Z}, j \in \mathbb{Z}\,.\, 0 \leq i < j < len(\zeta(p,t)) \implies \exists \tau_1, \tau_2 \in \mathbb{R}\,.\, \tau_1 < \tau_2 \wedge \langle \langle p, t, \zeta(p,t)[i] \rangle, \tau_1 \rangle \in C \wedge \langle \langle p, t, \zeta(p,t)[j] \rangle, \tau_2 \rangle \in C$
2. $\forall \langle \langle p, t, m_1 \rangle, \tau_1 \rangle, \langle \langle p, t, m_2 \rangle, \tau_2 \rangle \in C\,.\, \tau_1 < \tau_2 \implies \exists i, j \in \mathbb{Z}\,.\, i < j \wedge \zeta(p,t)[i] = m_1 \wedge \zeta(p,t)[j] = m_2$.

Definition 3 (Execution Projections). *We denote with $E^{DT}\big|_{\langle \sigma, C \rangle}$ the projection of an execution E^{DT} in the calendar model on the components σ and C, and with $E^{NT}\big|_{\langle \sigma, \zeta \rangle}$ the projection of an execution E^{NT} in the timeless model on the components σ and ζ. Furthermore, $E^{DT}\big|^{u}_{\langle \sigma, C \rangle}$ and $E^{NT}\big|^{v}_{\langle \sigma, \zeta \rangle}$ denote executions of lengths u and v in the corresponding models.*

Definition 4 (Stuttering Equivalence). *Two executions E^{DT} and E^{NT} are stuttering equivalent with respect to σ, C, and ζ, denoted by $E^{DT}\big|_{\langle \sigma, C \rangle} \sim E^{NT}\big|_{\langle \sigma, \zeta \rangle}$, if and only if:*

1. *The number of updates to σ and C/ζ (in terms of state and C-ζ equivalence) is the same in both executions;*
2. *The corresponding updates in both executions are equivalent.*

$$\textsc{Deliver}\quad \frac{t \in Topics \quad p \in t[subs] \quad len \circ \zeta(p,t) > 0 \quad m = \zeta(p,t)[0] \quad q = \kappa(p,t) \quad len \circ q < size(p,t)}{\langle S, \langle \mu, \kappa, lost\rangle, \zeta\rangle \xrightarrow[NT]{} \langle S, \langle \mu, \kappa', lost\rangle, \zeta'\rangle \quad \kappa' = \kappa[(p,t) \mapsto q \oplus m] \quad \zeta' = \zeta[(p,t) \mapsto \zeta(p,t)[1{:}]]}$$

$$\textsc{Deliver-Loss}\quad \frac{t \in Topics \quad p \in t[subs] \quad len \circ \zeta(p,t) > 0 \quad m = \zeta(p,t)[0] \quad q = \kappa(p,t) \quad len \circ q = size(p,t) \quad l = lost(p,t)}{\langle S, \langle \mu, \kappa, lost\rangle, \zeta\rangle \xrightarrow[NT]{} \langle S, \langle \mu, \kappa', lost'\rangle, \zeta'\rangle \quad \kappa' = \kappa[(p,t) \mapsto q[1{:}] \oplus m] \quad \zeta' = \zeta[(p,t) \mapsto \zeta(p,t)[1{:}]] \quad lost' = lost[(p,t) \mapsto l+1]}$$

$$\textsc{Activate}\quad \frac{p \in Enabled_{NT}}{\langle S[p \mapsto \varepsilon], \langle \mu, \kappa, lost\rangle, \zeta\rangle \xrightarrow[NT]{} \langle S[p \mapsto S_p], \langle \mu', \kappa', lost'\rangle, \zeta\rangle \quad \mu' = \mu[q_t \mapsto \kappa(p,t)] \text{ for every } t \in p[subscribe] \quad \kappa' = \kappa[(p,t) \mapsto [\,]] \text{ for every } t \in p[subscribe] \quad lost' = lost[(p,t) \mapsto 0] \text{ for every } t \in p[subscribe]}$$

$$\textsc{Read}\quad \frac{m \in Variables|_p \quad q = \mu(q_t) \quad q \neq [\,]}{\langle S[p \mapsto \text{read } m := tid; stmt], \langle \mu, \kappa, lost\rangle, \zeta\rangle \xrightarrow[NT]{} \langle S[p \mapsto stmt], \langle \mu', \kappa, lost\rangle, \zeta\rangle \quad \mu' = \mu[m \mapsto q_t[0], q_t \mapsto q_t[1{:}]]}$$

$$\textsc{Read-Empty}\quad \frac{m \in Variables|_p \quad \mu(q_t) = [\,]}{\langle S[p \mapsto \text{read } m := tid; stmt], \langle \mu, \kappa, lost\rangle, \zeta\rangle \xrightarrow[NT]{} \langle S[p \mapsto stmt], \langle \mu', \kappa, lost\rangle, \zeta\rangle \quad \mu' = \mu[m \mapsto null]}$$

$$\textsc{Publish}\quad \frac{Publishable(t, \kappa, \zeta, lost)}{\langle S[p \mapsto \text{publish}(t,m); stmt], \sigma, \zeta\rangle \xrightarrow[NT]{} \langle S[p \mapsto stmt], \sigma, \zeta'\rangle \quad \zeta' = \zeta[(p',t) \mapsto \zeta(p',t) \oplus m] \text{ for every } p' \in t[subs]}$$

$$\textsc{Reset}\quad \frac{}{\langle S[p \mapsto \text{return}], \sigma, \zeta\rangle \xrightarrow[NT]{} \langle S[p \mapsto \varepsilon], \sigma, \zeta\rangle}$$

Figure 8: Operational semantics of the timeless transition relation $\xrightarrow[NT]{}$.

Theorem 1. *For every execution E^{DT} in the calendar model, there exists an execution E^{NT} in the timeless model such that $E^{DT}|_{\langle\sigma,C\rangle} \sim E^{NT}|_{\langle\sigma,\zeta\rangle}$.*

Proof. We prove the theorem by induction on the execution length.

Base Case: We assume that $E^{DT}|^1_{\langle\sigma,C\rangle} = \langle\sigma_1^{DT}, C_1\rangle$ and $E^{NT}|^1_{\langle\sigma,\zeta\rangle} = \langle\sigma_1^{NT}, \zeta_1\rangle$, where $\sigma_1^{DT} \equiv \sigma_1^{NT}$ and $C_1 \equiv \zeta_1$. Hence, $E^{DT}|^1_{\langle\sigma,C\rangle} \sim E^{NT}|^1_{\langle\sigma,\zeta\rangle}$.

Induction Hypothesis: We assume that $E^{DT}|^u_{\langle\sigma,C\rangle} \sim E^{NT}|^v_{\langle\sigma,\zeta\rangle}$ for $u, v \in \mathbb{N}$, where

$$E^{DT}|^u_{\langle\sigma,C\rangle} = \langle\sigma_1^{DT}, C_1\rangle \ldots \langle\sigma_u^{DT}, C_u\rangle \text{ with } \sigma_u^{DT} = \langle\mu_u^{DT}, \kappa_u^{DT}, lost_u^{DT}\rangle \text{ and}$$
$$E^{NT}|^v_{\langle\sigma,\zeta\rangle} = \langle\sigma_1^{NT}, \zeta_1\rangle \ldots \langle\sigma_v^{NT}, \zeta_v\rangle \text{ with } \sigma_v^{NT} = \langle\mu_v^{NT}, \kappa_v^{NT}, lost_v^{NT}\rangle.$$

Induction step: We prove that $E^{DT}|^{u+1}_{\langle\sigma,C\rangle} \sim E^{NT}|^{v+k}_{\langle\sigma,\zeta\rangle}$ for $k \in \{0,1\}$, where

$$E^{DT}|^{u+1}_{\langle\sigma,C\rangle} = \langle\sigma_1^{DT}, C_1\rangle \ldots \langle\sigma_{u+1}^{DT}, C_{u+1}\rangle \text{ with } \sigma_{u+1}^{DT} = \langle\mu_{u+1}^{DT}, \kappa_{u+1}^{DT}, lost_{u+1}^{DT}\rangle \text{ and}$$
$$E^{NT}|^{v+k}_{\langle\sigma,\zeta\rangle} = \langle\sigma_1^{NT}, \zeta_1\rangle \ldots \langle\sigma_{v+k}^{NT}, \zeta_{v+k}\rangle \text{ with } \sigma_{v+k}^{NT} = \langle\mu_{v+k}^{NT}, \kappa_{v+k}^{NT}, lost_{v+k}^{NT}\rangle.$$

We proceed by showing for every transition rule (1) if the rule can trigger in the u^{th} calendar model configuration, then its corresponding rule can trigger in the v^{th} timeless model configuration as well; (2) triggering the corresponding rules in both models preserves the stuttering equivalence relation between executions.

Delay This rule does not exist in the timeless model, and it only updates the current time in the calendar model while leaving the program state and calendar unchanged. Therefore, it preserves stuttering equivalence as $E^{DT}|^{u+1}_{\langle\sigma,C\rangle} \sim E^{NT}|^{v}_{\langle\sigma,\zeta\rangle}$.

Deliver If the rule can trigger in the calendar model, there is a message delivery event $\langle\langle p,t,m\rangle, ct\rangle \in C_u$ and $len \circ \kappa_u^{DT}(p,t) < size(p,t)$. Hence, m has already been published to t and is the oldest message that is going to be delivered by p. Then, the induction hypothesis implies $m \in \zeta_v(p,t)$ and $len \circ \kappa_v^{NT}(p,t) < size(p,t)$. In addition, since m is the oldest message on t waiting to be delivered by p, it is also the first message in $\zeta_v(p,t)$. Therefore, m on topic t can be delivered by p without dropping a message in the timeless model as well. The rule in the calendar model removes $\langle\langle p,t,m\rangle, ct\rangle$ from C_u and inserts m into $\kappa_u^{DT}(p,t)$. In the timeless model, it removes m from $\zeta_v(p,t)$ and inserts it into $\kappa_v^{NT}(p,t)$. Therefore, $\kappa_{u+1}^{DT} = \kappa_{v+1}^{NT}$ and since μ_u^{DT}, μ_v^{NT}, $lost_u^{DT}$, and $lost_v^{NT}$ remain unchanged, $\sigma_{u+1}^{DT} \equiv \sigma_{v+1}^{NT}$. In addition, removing m from $\zeta_v(p,t)$ and its corresponding event from C_u ensures that $C_{u+1} \equiv \zeta_{v+1}$, and thereby $E^{DT}|^{u+1}_{\langle\sigma,C\rangle} \sim E^{NT}|^{v+1}_{\langle\sigma,\zeta\rangle}$.

Deliver-Loss If the rule can trigger in the calendar model, there is a message delivery event $\langle\langle p,t,m\rangle, ct\rangle \in C_u$ and $len \circ \kappa_u^{DT}(p,t) = size(p,t)$. By following the same line of reasoning as for Deliver, the induction hypothesis implies $m \in \zeta_v(p,t)$ and $len \circ \kappa_v^{NT}(p,t) = size(p,t)$. In addition, m is the first message in $\zeta_v(p,t)$. Therefore, m on topic t can be delivered by p and the oldest message dropped from the receiving buffer in the timeless model as well. The rule in the calendar model removes $\langle\langle p,t,m\rangle, ct\rangle$ from C_u, inserts m into $\kappa_u^{DT}(p,t)$ by removing the oldest message, and increments $lost_u^{DT}(p,t)$. In the timeless model, it removes m from $\zeta_v(p,t)$, inserts it into $\kappa_v^{NT}(p,t)$ by removing the oldest message, and increments $lost_v^{NT}(p,t)$. Therefore, $\kappa_{u+1}^{DT} = \kappa_{v+1}^{NT}$, $lost_{u+1}^{DT} = lost_{v+1}^{NT}$, and since μ_u^{DT} and

μ_v^{NT} remain unchanged, $\sigma_{u+1}^{DT} \equiv \sigma_{v+1}^{NT}$. Similarly to Deliver, $C_{u+1} \equiv \zeta_{v+1}$, and thereby $E^{DT}|_{\langle\sigma,C\rangle}^{u+1} \sim E^{NT}|_{\langle\sigma,\zeta\rangle}^{v+1}$.

Activate If the rule can trigger in the calendar model, then there is a process $p \in Enabled_{DT}$. Since the modeled QPDS satisfies our constraint satisfaction problem, process p has the required minimum number of messages in its receiving buffers when it is activated (see Constraint (3) in Section 4.1). Then, the induction hypothesis implies that for every topic t to which p subscribes, $len \circ \kappa_v^{NT}(p,t) \geq min\_new(p,t)$ in the timeless model, meaning that $p \in Enabled_{NT}$. Hence, the rule can trigger in the timeless model as well. The rule in both models only moves the contents of receiving buffers into local buffers, i.e., it updates σ_u^{DT} and σ_v^{NT} in the same way, and thus $\sigma_{u+1}^{DT} \equiv \sigma_{v+1}^{NT}$. In addition, C_u in the calendar model and ζ_v in the timeless model remain unchanged, so $C_{u+1} \equiv \zeta_{v+1}$. Hence, $E^{DT}|_{\langle\sigma,C\rangle}^{u+1} \sim E^{NT}|_{\langle\sigma,\zeta\rangle}^{v+1}$.

Read, Read-Empty If rule Read can trigger in the calendar model, $\mu_u^{DT}(q_t)$ is a non-empty sequence of messages. Since the induction hypothesis implies $\mu_v^{NT}(q_t)$ is also a non-empty sequence of messages, the rule can trigger in the timeless model as well. Rule Read removes the first message from $\mu_u^{DT}(q_t)$ and $\mu_v^{NT}(q_t)$ and assigns it to a local variable m in the corresponding models. According to the induction hypothesis, the first elements in $\mu_u^{DT}(q_t)$ and $\mu_v^{NT}(q_t)$ are equal. Hence, $\mu_{u+1}^{DT}(q_t) = \mu_{v+1}^{NT}(q_t)$ and $\mu_{u+1}^{DT}(m) = \mu_{v+1}^{NT}(m)$. In addition, C_u, κ_u^{DT}, $lost_u^{DT}$ in the calendar model and ζ_v, κ_v^{NT}, $lost_v^{NT}$ in the timeless model remain unchanged. We conclude that $E^{DT}|_{\langle\sigma,C\rangle}^{u+1} \sim E^{NT}|_{\langle\sigma,\zeta\rangle}^{v+1}$. Using the same line of reasoning, we can prove this for rule Read-Empty as well.

Publish Suppose the rule can trigger in the calendar model such that process p publishes m to topic t. Our constraint satisfaction problem guarantees that for every process p' that subscribes to t, the number of possible lost messages $lost_u^{DT}$ does not exceed $max\_loss(p',t)$. Therefore, the number of messages published to t for every subscriber p', which is equal to the sum of the number of messages in $\kappa_u^{DT}(p',t)$, message delivery events for p' through t in C_u, and $lost_u^{DT}(p',t)$ is less than $size(p',t) + max\_loss(p',t)$. In addition, the induction hypothesis implies that the number of published messages in the calendar and timeless model is the same, since otherwise either $C_u \not\equiv \zeta_v$, $\kappa_u^{DT}(p',t) \neq \kappa_v^{NT}(p',t)$, or $lost_u^{DT} \neq lost_v^{NT}$. Hence, we conclude that the number of published messages in the timeless model, which is $len \circ \kappa_v^{NT}(p',t) + len \circ \zeta_v(p',t) + lost_v^{NT}(p',t)$, is less than $size(p',t) + max\_loss(p',t)$ as well. Then, as defined earlier, $Publishable(t, \kappa_v^{NT}, \zeta_v, lost_v^{NT})$ holds, and the rule can trigger in the timeless model as well. In the calendar model, if the rule triggers for a process p that publishes message m to topic t, it adds $\langle\langle p',t,m\rangle,\tau\rangle$, where $\tau \geq ct$, to C_u for every p' that subscribes to t. In the timeless model, the rule inserts m into $\zeta_v(p',t)$ for every p' that subscribes to t, and thus $C_{u+1} \equiv \zeta_{v+1}$. Furthermore, the rule leaves the program state unchanged in both models. Hence, we conclude that $E^{DT}|_{\langle\sigma,C\rangle}^{u+1} \sim E^{NT}|_{\langle\sigma,\zeta\rangle}^{v+1}$.

Reset If the rule can trigger in the calendar model, it can obviously trigger in the timeless model as well, as its premise is *true*. In the calendar model, the rule leaves the program state unchanged and it does not add (resp. remove) message delivery events to (resp. from) the calendar. In the timeless model, the rule leaves the program state and communication channels unchanged. That concludes $\sigma_{u+1}^{DT} \equiv \sigma_{v+1}^{NT}$ and $C_{u+1} \equiv \zeta_{v+1}$ since according to Definition 2 we do not take process activation events in the calendar into account when determining this equivalence. Hence, $E^{DT}|_{\langle\sigma,C\rangle}^{u+1} \sim E^{NT}|_{\langle\sigma,\zeta\rangle}^{v+1}$.

□

Completeness Our timeless model allows executions that the calendar model does not. Hence, it could introduce false alarms for safety properties, and it is also not suitable for checking of liveness properties. The most obvious situation that introduces such executions is when, in the calendar model, a subscriber process p to a topic t is activated more often than its publisher p', and t is the only topic it is subscribed to. In that case, $min\_new(p,t) = 0$, which results in p being always enabled in the timeless model. For example, the *Actuator* process in our example in Section 2.1 is always enabled in the timeless model since $min\_new(Actuator, Power) = 0$. One such execution that exists in the timeless but not in the calendar model is: *Sensor*, *Sensor*, *Actuator*, *Actuator*, *Actuator*, *Sensor*, *Sensor*, *Actuator*, *Sensor*, *Controller*, *Actuator* .... We conjecture that such executions would rarely introduce false alarms since such processes must be implemented to be able to handle the case when zero messages are present at their activation.

6 IMPLEMENTATION AND EVALUATION

Our prototype implementation uses SMT solver Z3 [14] to solve the constraint satisfaction problems we generate (see Figure 5) and SPIN [24] to perform model checking. We developed several benchmarks to evaluate our proposed timeless model.

Pilot Flying System consists of two left and right Flight Guidance Systems (FGS), which need to agree on which side is the pilot flying side of the aircraft. A pilot may transfer the control between sides at any time. We implemented the timeless model of this example based on its previously published quasi-synchronous model [30]. A FGS compares the measured position, speed, and altitude of an aircraft with the desired state, and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The two redundant FGSs communicate through a cross-channel bus. We implement each FGS as a process, where two redundant sides communicate through delaying topics. We check the property that at least one side is always the pilot flying side (at_least_one_side).

Ground Vehicle is the example we introduced in Section 2.1. The controller process maintains the speed of the vehicle in a valid range by controlling the power of the engine. We check the property that the speed of the vehicle remains in the valid range if there is no obstacle and the operator does not push the stop button (speed_in_range).

House Thermostat ensures the room temperature is within a preset range. In a home thermostat controller, a thermometer

Table 1: Experimental results. Column *Storage Mode* is the mode used for storing of the visited states; *#P* is the number of processes in the SPIN model; *#Stored* is the number of states stored; *#Matched* is the number of states matched; *Memory* is the total memory consumption of model checking; *Time* is the verification time.

Benchmark	Property	SPIN						AGREE
		Storage Mode	#P	#Stored	#Matched	Memory (MB)	Time (s)	Time (s)
Pilot Flying System	at_least_one_side	exhaustive	4	8819899	30143418	1352	10	276
Ground Vehicle	speed_in_range	bitstate	5	3918658	12486754	763	148	—
House Thermostat	temperature_in_range	bitstate	6	3936344	11547677	1957	155	—

measures the room temperature, based on which the thermostat regulates the temperature by controlling the heater. We check the property that if a user does not change the temperature, it remains within the preset range (temperature_in_range).

Table 1 shows the verification results for our benchmarks. We performed our experiments on a 4.00 GHz Intel Core i7 Linux machine with 62 GB of memory; note that the SPIN model checker uses only one core. We use the pilot flying system benchmark to compare the performance of our timeless model we implemented in SPIN with AGREE [1] as published in Miller et al. [30]. AGREE [1] model checker uses Kind [27] or JKind [25] (a lighter version of Kind) to perform k-induction. Since we failed to install AGREE on our machine in order to perform a completely fair comparison [20], we report the running times of AGREE from the paper [30], which were obtained on a slower machine. This is also the reason why we did not implement the ground vehicle and house thermostat benchmarks in AGREE. We model checked all benchmarks in less than 3 minutes each. These preliminary results show the promise of our timeless model since it is (at least) comparable to a state-of-the-art tool (AGREE).

Due to a large number of processes and topics, SPIN cannot complete the exploration of search space of the ground vehicle and house thermostat benchmarks. Hence, we model check them with bitstate hashing enabled [23]. Bitstate hashing is a well-known technique that tries to improve the state space coverage using a hashtable. A hash function maps a program state to an entry in the hashtable, which is marked if the corresponding state is visited. The technique may cause bugs to be missed, but when using a large memory for the bitstate space and several hash functions, it typically covers a large portion of the state space (99% or more).

Due to the blocking publish in SPIN, deadlocks can occur in the timeless model. That happens when a process is trying to publish more messages than it is allowed. Such executions are infeasible in the calendar model due to tracking of time and timing constraints of a QPDS, and they do not expose quasi-periodic behavior. Therefore, we can safely recover from such deadlocks by enforcing that one of the processes with the maximum number of messages in its inbox skips the rest of its current step and continues the execution. Since we only address deadlocks in non-quasi-periodic executions, we do not interfere with the proof that the timeless model is an overapproximation of the calendar model.

7 RELATED WORK

In the most closely related work [28], Larrieu and Shankar propose a quasi-synchronous model for QPDSs, which is inspired by the Robot Operating System (ROS). They also prove the soundness properties of their model. Based on this computational model, Li et al. [29] more recently present the RADL framework for designing and verifying QPDSs. They also provide an encoding of such systems using calendar automata in the Symbolic Analysis Laboratory (SAL) [6]. Then, Baudart et al. [3, 4] show that the quasi-synchronous abstraction, introduced by Caspi [9–11], is not sound for general systems of more than two processes. They propose conditions on the communication topologies and timing parameters to recover soundness. In our work, we leverage the soundness properties of the computational model proposed by Larrieu and Shankar and the soundness recovery conditions proposed by Baudart et al. to build our constraint satisfaction problem and design our timeless model (see Section 3.1). All quasi-synchronous models, such as the calendar model, have a discrete notion of time, while our finite-state model completely eliminates it by instead using buffer predicates for process synchronization.

Saha et al. [35] present a finitary reduction technique to formalize clockless finite-state timeout and calendar-based models. They consider the timeout value is an integer between zero and the maximum of the upper bounds of timeouts for all processes. Although this work reduces the infinite-state of the calendar model to finite-state, it still uses a discrete-time model and addresses only general real-time systems. On the other hand, we propose a customized timeless model for QPDSs. Roy et al. [34] formalize the timeout-based clockless model and prove that it simulates the timeout-based real-time model. They claim that using a similar approach one can also prove that the calendar-based clockless model simulates the calendar-based real-time model. Loosely Time-Triggered Architecture (LTTA) [7, 36] is another abstract model for QPDSs, which uses shared memory for communication between processes and models delays as part of process periods. Unlike our timeless model, LTTA models clocks as Boolean variables; it also assumes that the memory is shared between all processes.

Bhattacharyya et al. [8] and Miller et al. [30] present a translator from a subset of SysML, which is a common system architecture modeling language, into the Architectural Analysis and Description Language (AADL). They also provide translators from AADL models into the input language of the UPPAAL [5] and Kind [27] model checkers. UPPAAL model checks timed-automata-based models, while Kind is a k-induction-based model checker. To

verify quasi-synchronous systems in Kind, they define an *acceptor automaton* in Lustre [21] that ensures quasi-synchrony. Halbwachs and Mandel [22] model and validate quasi-synchronous processes using synchrony. They model an acceptor automaton of two quasi-synchronous processes and a scheduler based on the acceptor, which generates process activation conditions for those two processes. These approaches leverage discrete clocks and acceptor automata to ensure the quasi-synchrony requirements, while our work totally eliminates time from the model.

Obermaisser et al. [31] extend finite state machines to include the *sparse global time base* (if events can occur in only some section of the timeline, the time base is said to be sparse), which is called *Periodic Finite State Machines* (PFSM). Then, the PFSM model includes global time, periodic clock constraints, and time-triggered activities as well, and it is useful in modeling distributed real-time systems. The specification of an application as a set of PFSMs is verified using SAL, which accepts discrete transition systems. PFSMs allow users to model and verify real-time systems as discrete-time models, whereas our work considers QPDSs, a specific kind of real-time systems, and allows users to model them without time.

N-synchrony [13] is another relaxed clock equivalence principle, which bounds the difference between the cumulative process activation counts. Desai et al. [17] formalize a similar model, called approximate synchrony, and compute this bound using parameters of a real-time system. This is unlike quasi-synchrony, which bounds drifts, and hence also the number of process activations between two consecutive activations of another process. Their work synchronizes processes by applying a constant delay to one of them and introducing a global buffer of size N. DRONA [16] is a software framework built on top of the P language [15] for developing reliable distributed mobile robotic systems; it also leverages approximate synchrony.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a simple domain-specific language for QPDSs, and formalized the operational semantics of the calendar model, which had been proposed by others for the verification such systems. Then, we defined a constraint satisfaction problem to check whether a QPDS is amenable to sound modeling using the user-provided parameters of the system. We use it to formalize the operational semantics of the timeless model we proposed; we also proved that the timeless model is an overapproximation of the calendar model. To the best of our knowledge, our model is the first finite-state timeless model for QPDSs. Since it completely eliminates time from the model, it is amenable to verification using model checkers for finite-state systems. We prototyped our approach using the SPIN model checker, and our preliminary experimental results show its promise in reducing the verification runtime. We plan to leverage property P4 in Section 3.1 to devise a complete model, and thereby enable the verification of liveness properties. We expect this to be feasible at the cost of enlarging the program state.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (NSF) awards CCF 1552975 and CCF 1837051. We thank Ankush Desai for introducing us to the problem of the verification of quasi-periodic distributed systems. We also thank Natarajan Shankar for insightful discussions that helped us to improve this paper.

REFERENCES

[1] agree [n.d.]. Assume Guarantee Reasoning Environment. http://loonwerks.com/tools/agree.html.
[2] Rajeev Alur and David L. Dill. 1994. A Theory of Timed Automata. *Theoretical Computer Science* 126, 2 (1994), 183–235. https://doi.org/10.1016/0304-3975(94)90010-8
[3] Guillaume Baudart. 2017. *Synchronous Approach to Quasi-Periodic Systems*. Ph.D. Dissertation. PSL Research University.
[4] Guillaume Baudart, Timothy Bourke, and Marc Pouzet. 2016. Soundness of the Quasi-synchronous Abstraction. In *Proceedings of the 16th International Conference on Formal Methods in Computer-Aided Design*. 9–16. https://doi.org/10.1109/FMCAD.2016.7886655
[5] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1996. UPPAAL — a Tool Suite for Automatic Verification of Real-time Systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III : Verification and Control*. 232–243. https://doi.org/10.1007/BFb0020949
[6] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. 2000. An Overview of SAL. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*. 187–196.
[7] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. 2002. A Protocol for Loosely Time-Triggered Architectures. In *Proceedings of the 2nd International Conference on Embedded Software*. 252–265. https://doi.org/10.1007/3-540-45828-X_19
[8] S. Bhattacharyya, S. Miller, J. Yang, S. Smolka, B. Meng, C. Sticksel, and C. Tinelli. 2014. Verification of quasi-synchronous systems with UPPAAL. In *Proceedings of the 33rd Digital Avionics Systems Conference*. 8A4–1–8A4–12. https://doi.org/10.1109/DASC.2014.6979532
[9] Paul Caspi. 2000. *The quasi-synchronous approach to distributed control systems*. Technical Report. Verimag, Crysis Project.
[10] Paul Caspi. 2001. Embedded Control: From Asynchrony to Synchrony and Back. In *Proceedings of the 1st International Workshop on Embedded Software*. 80–96. https://doi.org/10.1007/3-540-45449-7_7
[11] Paul Caspi, Christine Mazuet, and Natacha Reynaud Paligot. 2001. About the Design of Distributed Control Systems: The Quasi-Synchronous Approach. In *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security*. 215–226. https://doi.org/10.1007/3-540-45416-0_21
[12] Edmund M. Clarke and E. Allen Emerson. 1982. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Proceedings of the Workshop on Logic of Programs*. 52–71. https://doi.org/10.1007/BFb0025774
[13] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. 2006. N-synchronous Kahn Networks: A Relaxed Model of Synchrony for Real-time Systems. In *Proceedings of the 33rd Symposium on Principles of Programming Languages*. 180–193. https://doi.org/10.1145/1111320.1111054
[14] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
[15] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-driven Programming. In *Proceedings of the 34th International Conference on Programming Language Design and Implementation*. 321–332. https://doi.org/10.1145/2491956.2462184
[16] Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A. Seshia. 2017. DRONA: A Framework for Safe Distributed Mobile Robotics. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*. 239–248. https://doi.org/10.1145/3055004.3055022
[17] Ankush Desai, Sanjit A. Seshia, Shaz Qadeer, David Broman, and John C. Eidson. 2015. Approximate Synchrony: An Abstraction for Distributed Almost-Synchronous Systems. In *Proceedings of the 27th International Conference on Computer Aided Verification*. 429–448. https://doi.org/10.1007/978-3-319-21668-3_25
[18] Bruno Dutertre and Maria Sorea. 2004. Modeling and Verification of a Fault-Tolerant Real-Time Startup Protocol Using Calendar Automata. In *Proceedings of the 8th International Symposium on Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. 199–214. https://doi.org/10.1007/978-3-540-30206-3_15
[19] Bruno Dutertre and Maria Sorea. 2004. *Timed Systems in SAL*. Technical Report. SRI International.
[20] githubissue [n.d.]. GitHub Issue We Reported About Failing AGREE Installation. https://github.com/loonwerks/formal-methods-workbench/issues/24.
[21] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

https://doi.org/10.1109/5.97300

[22] Nicolas Halbwachs and Louis Mandel. 2006. Simulation and Verification of Asynchronous Systems by Means of a Synchronous Model. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*. 3–14. https://doi.org/10.1109/ACSD.2006.24

[23] Gerard J. Holzmann. 1998. An Analysis of Bitstate Hashing. *Formal Methods in System Design* 13, 3 (1998), 289–307. https://doi.org/10.1023/A:1008696026254

[24] Gerard J. Holzmann. 2004. *The Spin Model Checker: Primer and Reference Manual* (1 ed.). Addison-Wesley.

[25] jkind [n.d.]. An infinite-state model checker for safety properties. http://loonwerks.com/tools/jkind.html.

[26] Neil D. Jones. 1997. *Computability and Complexity: From a Programming Perspective*. MIT Press.

[27] Temesghen Kahsai and Cesare Tinelli. 2011. PKIND: A parallel k-induction based model checker. In *Proceedings of the 10th International Workshop on Parallel and Distributed Methods in Verification (EPTCS)*, Vol. 72. 55–62. https://doi.org/10.4204/EPTCS.72.6

[28] R. Larrieu and N. Shankar. 2014. A framework for high-assurance quasi-synchronous systems. In *Proceedings of the 12th International Conference on Formal Methods and Models for Codesign*. 72–83. https://doi.org/10.1109/MEMCOD.2014.6961845

[29] W. Li, L. Gérard, and N. Shankar. 2015. Design and verification of multi-rate distributed systems. In *Proceedings of the 13th International Conference on Formal Methods and Models for Codesign*. 20–29. https://doi.org/10.1109/MEMCOD.2015.7340463

[30] Steven P. Miller, Sidhartha Bhattacharyya, Cesare Tinelli, Scott Smolka, Christoph Sticksel, Baoluo Meng, and Junxing Yang. 2015. *FORMAL VERIFICATION OF QUASI-SYNCHRONOUS SYSTEMS*. Technical Report. Rockwell Collins.

[31] R. Obermaisser, C. El-Salloum, B. Huber, and H. Kopetz. 2007. Modeling and verification of distributed real-time systems using periodic finite state machines. *Computer Systems Science and Engineering* 22, 6 (2007).

[32] Jean-Pierre Queille and Joseph Sifakis. 1982. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the International Symposium on Programming*. 337–351. https://doi.org/10.1007/3-540-11494-7_22

[33] ros [n.d.]. Robot Operating System (ROS). http://www.ros.org.

[34] Suman Roy, Janardan Misra, and Indranil Saha. 2016. A Simplification of a Real-time Verification Problem. *Software Testing, Verification and Reliability* 26, 8 (2016), 548–571. https://doi.org/10.1002/stvr.1622

[35] Indranil Saha, Janardan Misra, and Suman Roy. 2007. Timeout and Calendar Based Finite State Modeling and Verification of Real-time Systems. In *Proceedings of the 5th International Conference on Automated Technology for Verification and Analysis*. 284–299. https://doi.org/10.1007/978-3-540-75596-8_21

[36] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale. 2008. Implementing Synchronous Models on Loosely Time Triggered Architectures. *IEEE Trans. Comput.* 57, 10 (2008), 1300–1314. https://doi.org/10.1109/TC.2008.81