Quasi-synchrony

Timothy Bourke[1,2]

1. INRIA Paris

2. École normale supérieure (DI)

Timothy.Bourke@inria.fr

These slides describe research together with Guillaume Baudart and Marc Pouzet

19 November 2018, MPRI: Parallélisme synchrone

Outline

The synchronous language Lustre

[Caspi, Pilaud, Halbwachs, and Plaice (1987): LUSTRE: A declarative language for programming synchronous systems]

- Ideal for programming an important class of embedded controllers.
  - Academic foundation of Scade Suite tool for critical industrial systems.
- Based on a discrete-time abstraction.

```
every trigger:
  read inputs;
  compute;
  write outputs
```

model: R_1, R_2, R_3, R_4, R_5, ...

Try to ignore 'physical time'; ensure that WCET $<$ period.

The synchronous language Lustre

[Caspi, Pilaud, Halbwachs, and Plaice (1987): LUSTRE: A declarative language for programming synchronous systems]

- Ideal for programming an important class of embedded controllers.
  - » Academic foundation of Scade Suite tool for critical industrial systems.
- Based on a discrete-time abstraction.

```
every trigger:
  read inputs;
  compute;
  write outputs
```

model: R_1, R_2, R_3, R_4, R_5, ...

Try to ignore 'physical time'; ensure that WCET < period.

This lecture:

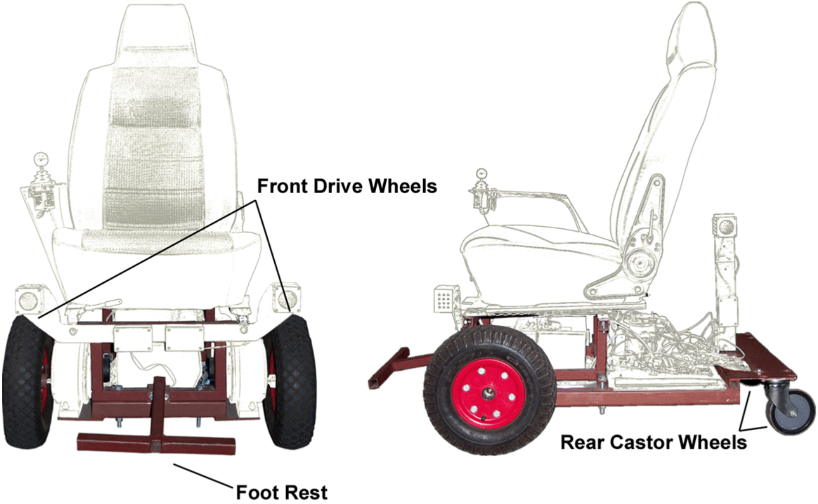
- Lustre: not just for programming, but also for modelling.
  - » Model, simulate, verify entire systems; programs in their environment.
  - » Formally model discrete systems (but with parallel composition, functional abstraction, etcetera).
- Introduce elements of real time for two reasons:
  1. Relate the discrete-time scales of concurrent programs.
  2. More naturally express the constraints of certain applications.

Outline

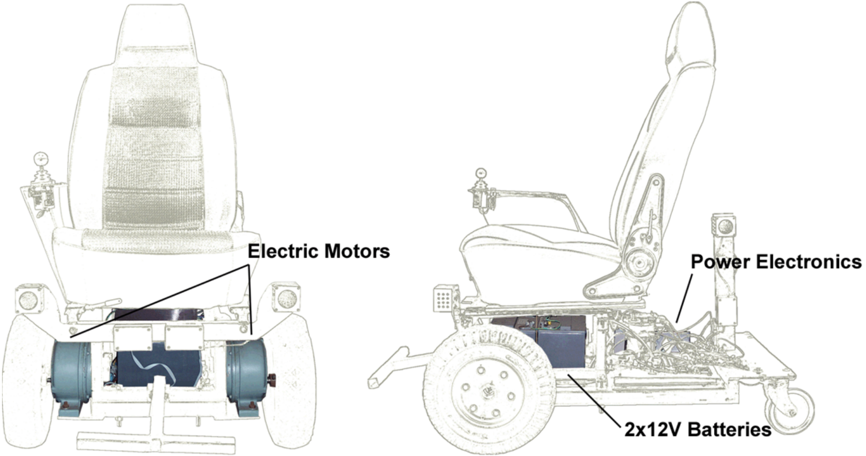
Wheelchair: An old, simple, but concrete example

- The UOW 'robotic' wheelchair
- Goal: low-cost mobility assistance
- Target of engineering student projects

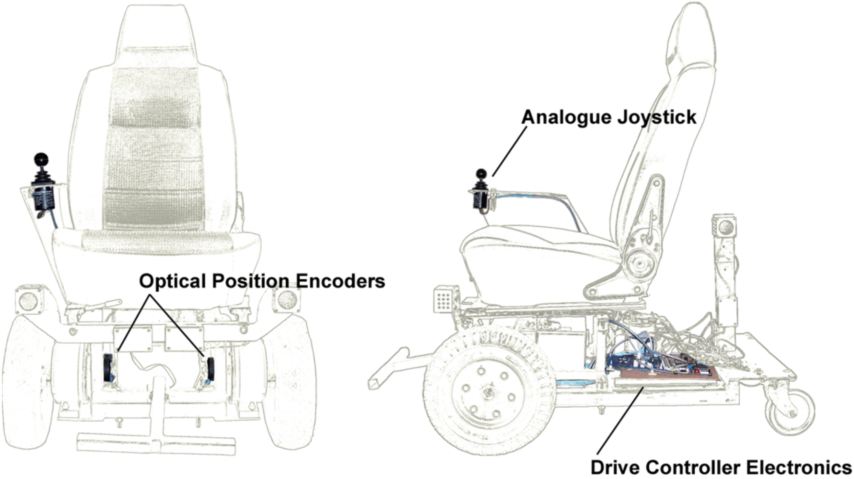
Wheelchair: mechanical structure



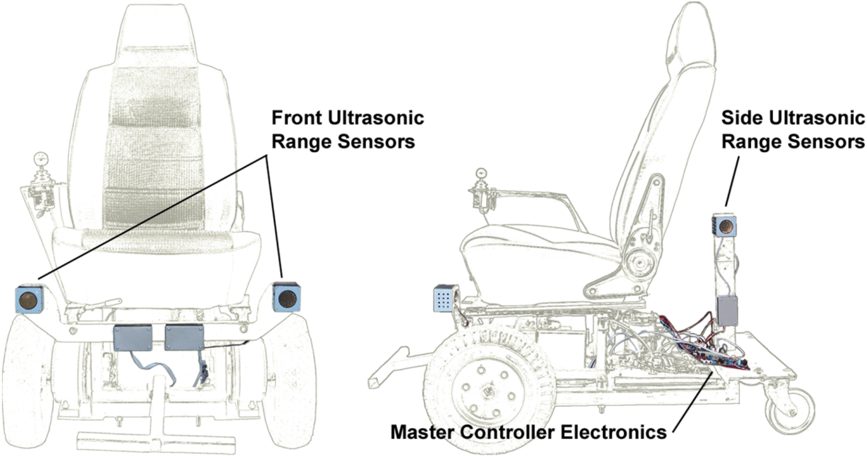
Wheelchair: power electronics



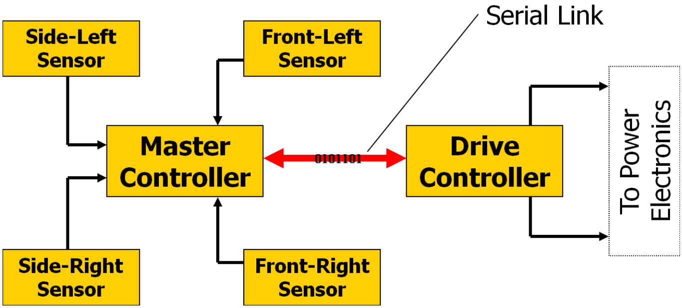
Wheelchair: driver controller



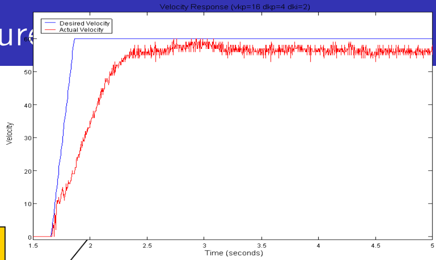
Wheelchair: master controller

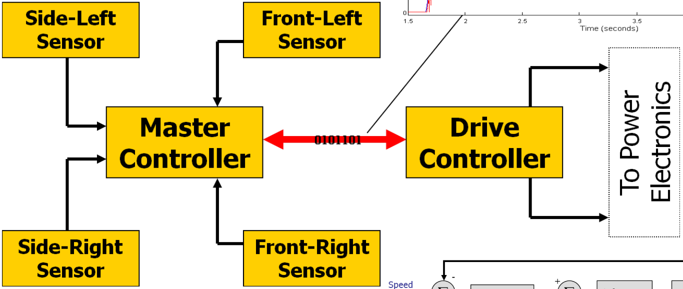


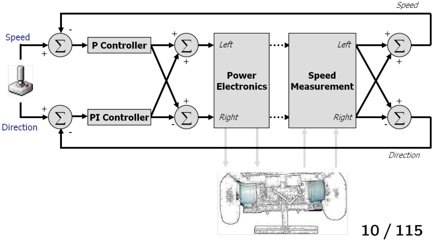
Wheelchair: architecture and functions



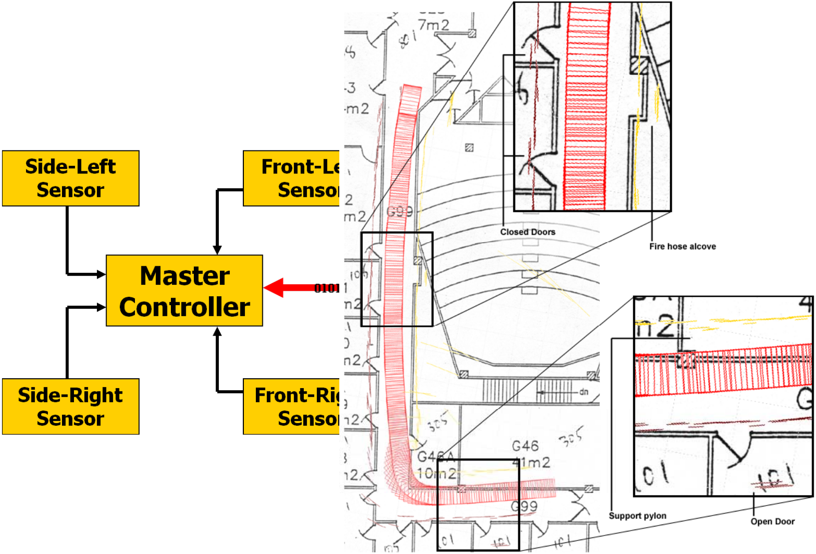
Wheelchair: architecture







Wheelchair: architecture and functions



System features

Not just old wheelchairs! Airbus command-control, automotive, et cetera.

Multiple processors...

- Running distinct tasks
  - Different timing characteristics
  - Different criticalities
  - Able to tolerate certain failures
  - Facilitate design and integration
- Possibly at different locations
  - For proximity to hardware
  - Due to system or operational constraints (e.g., maintenance)
  - For fault-tolerance

...communicating over:

- serial links,
- Fieldbus (IEC 61158),
- CAN networks,
- Ethernet (AFDX),
- et cetera.

System features

Not just old wheelchairs! Airbus command-control, automotive, et cetera.

Multiple processors. . .

- Running distinct tasks
  - Different timing characteristics
  - Different criticalities
  - Able to tolerate certain failures
  - Facilitate design and integration
- Possibly at different locations
  - For proximity to hardware
  - Due to system or operational constraints (e.g., maintenance)
  - For fault-tolerance

. . . communicating over:

- serial links,
- Fieldbus (IEC 61158),
- CAN networks,
- Ethernet (AFDX),
- et cetera.

Sometimes synchronized

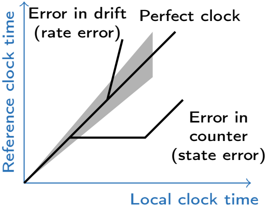
- Time-Triggered Protocol (TTP)
- Precision Time Protocol (PTP)

But not always

- Quasi-synchrony

Clocks[1]

- A (physical) clock k comprises
  - a *physical oscillation mechanism* generating microticks, and,
  - a *counter*.
- Reference clock z: observe all events, ignoring relativity
  - Frequency f^z completely agrees with international time standard, and
  - very large: 10^{15} *microticks*/*sec*.; granularity = 1 femtosecond ($10^{-15}\,\mathrm{s}$).



The drift rate of clock k:

$$\rho_k(i) = \left| \frac{z(microtick_k(i+1)) - z(microtick_k(i))}{n_k} - 1 \right|$$

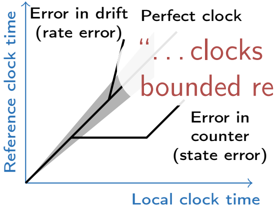
where n_k is the nominal number of reference clock ticks in a granule.

- Real clocks have varying drift rates influenced by, for e.g., changes in temperature or applied voltage, or aging of the crystal resonator.
- Bounded by a *maximum drift rate* $\rho_{k_{\max}}$.

[1] H. Kopetz (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Chapter 3

Clocks[1]

- A (physical) clock k comprises
  - a *physical oscillation mechanism* generating microticks, and,
  - a *counter*.
- Reference clock z: observe all events, ignoring relativity
  - Frequency f^z completely agrees with international time standard, and
  - very large: 10^{15}*microticks*/*sec*.; granularity = 1 femtosecond ($10^{-15}\,\mathrm{s}$).



The drift rate of clock k:

"…clocks that are never resynchronized leave any bounded relative time interval after a finite time…"

$$\rho_k(i) = \left| \frac{z(microtick_k(i+1)) - z(microtick_k(i))}{n_k} - 1 \right|$$

where n_k is the nominal number of reference clock ticks in a granule.

- Real clocks have varying drift rates influenced by, for e.g., changes in temperature or applied voltage, or aging of the crystal resonator.
- Bounded by a *maximum drift rate* $\rho_{k_{\max}}$.

[1] H. Kopetz (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Chapter 3

Clock drift risk

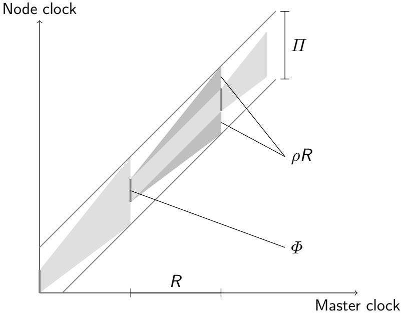
Kopetz[2] summarises Neumann[3] (my emphasis):

> ***Example:*** *During the Gulf war on February 25, 1991 a Patriot missile defense system failed to intercept an incoming scud rocket. The clock drift over a 100 hour period (which resulted in a tracking error of 678 meters) was blamed for the Patriot missing the scud missile that hit an American military barracks in Dhahran, killing 29 and injuring 97. The original requirement was a 14 hour mission. The clock drift during a 14 hour mission could be handled.*

[2] H. Kopetz (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. p.49

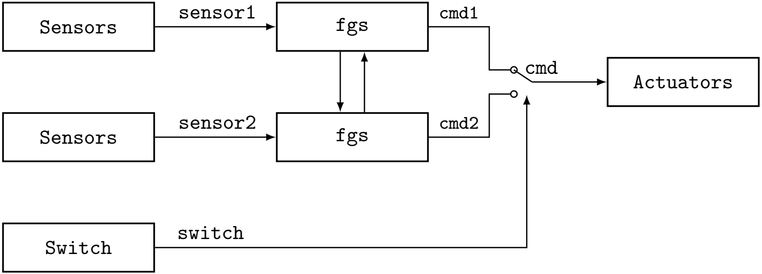
[3] P. G. Neumann (1994). *Computer Related Risks*. p.34

Central Master Clock Synchronization[4]



- Node clocks stay within the shaded area.
- R is the resynchronization interval.
- Φ is the offset after resynchronization.
- ρ is the drift rate between two clocks.
- Π is the protocol's precision.

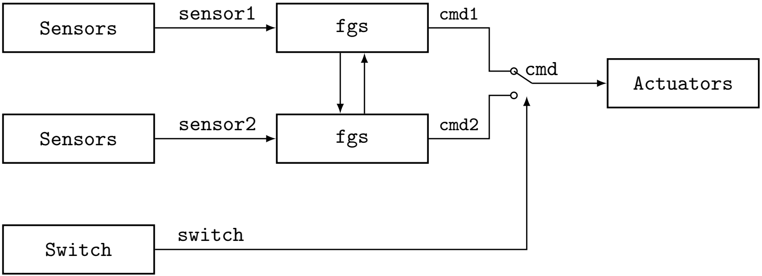
[4] H. Kopetz (2011). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd ed., Figure 3.10



Flight Guidance System

[Miller, Bhattacharyya, Tinelli, Smolka, Sticksel, Meng, and Yang (2015): Formal Verification of Quasi-Synchronous Systems]

- Periodically generates commands to control an aircraft's trajectory.
- Implemented in two redundant and *unsynchronized* modules.
- A manual transfer switch changes from one to the other.
- The components share information to avoid glitches during transfer.



Flight Guidance System

[Miller, Bhattacharyya, Tinelli, Smolka, Sticksel, Meng, and Yang (2015): Formal Verification of Quasi-Synchronous Systems]

```
let node controller (sensor1, sensor2, switch) = cmd where
  rec cmd1 = fgs(sensor1, idle fby cmd2)
  and cmd2 = fgs(sensor2, idle fby cmd1)
  and cmd = if switch then cmd1 else cmd2
```

val controller: data $\times$ data $\times$ bool $\xrightarrow{D}$ cmd

Quasi-periodic Architecture: intuitions

- Multiple processors running periodic tasks.
- Clocks are not synchronized.
- Communication: transmission with bounded delay to local memories with sampling upon activation.
- 'Blackboard' systems [Berry (1989): Real Time Programming: Special Purpose or General Purpose Languages]
- Assume reliable network: no message loss and preserves ordering.

Definition 1 (Quasi-periodic architecture)

A *quasi-periodic architecture* is a finite set of processors, or nodes $\mathcal{N}$, where

1. every node $n \in \mathcal{N}$ executes almost periodically, that is, the actual time between any two successive activations $T \in \mathbb{R}$ may vary between known bounds during an execution,

$$0 \leq T_{\min} \leq T \leq T_{\max}, \tag{RP}$$

2. values are transmitted between processes with a delay $\tau \in \mathbb{R}$, bounded by $\tau_{\min}$ and $\tau_{\max}$,

$$0 \leq \tau_{\min} \leq \tau \leq \tau_{\max}. \tag{RT}$$

Synchronous Real-Time Model

Definition 1 (Quasi-periodic architecture)

A *quasi-periodic architecture* is a finite set of processors, or nodes $\mathcal{N}$, where

1. every node $n \in \mathcal{N}$ executes almost periodically, that is, the actual time between any two successive activations $T \in \mathbb{R}$ may vary between known bounds during an execution,

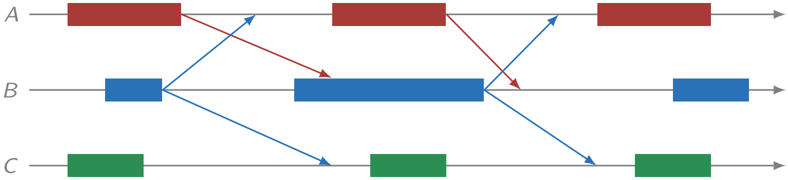
$$0 \leq T_{\min} \leq T \leq T_{\max}, \tag{RP}$$

2. values are transmitted between processes with a delay $\tau \in \mathbb{R}$, bounded by $\tau_{\min}$ and $\tau_{\max}$,

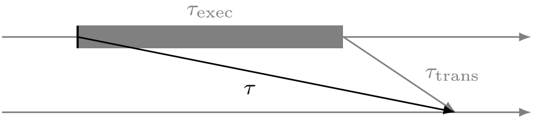
$$0 \leq \tau_{\min} \leq \tau \leq \tau_{\max}. \tag{RT}$$

Equivalently, a process is characterized by a nominal period T^{nom} and maximum jitter ε (variability of delay), where $0 \leq \varepsilon < T^{\text{nom}}$ ($T_P^{\min} = T_P^{\text{nom}} - \varepsilon_P$ and $T_P^{\max} = T_P^{\text{nom}} + \varepsilon_P$).

Real-time trace of a quasi-periodic architecture



Example real-time trace with three nodes.
Rectangles represent tasks, arrows denote message transmissions.
Note the jitter both on node activation periods and transmission delays.



Abstract tasks as instantaneous activations with a communication delay τ that encompasses both the execution time τ_{exec} and the transmission delay τ_{trans}: $\tau = \tau_{\text{exec}} + \tau_{\text{trans}}$

So what?

These assumptions are very general. They are not hard to satisfy.
They thus potentially apply to many systems.

Given a quasi-periodic architecture $(T^{\mathrm{nom}}, \varepsilon)$, we can

- Bound the delay between the generation and use of a value,
- Bound overwrites, the number of values lost due to undersampling,
- Bound oversamples, the number of times a single value is read,

So what?

These assumptions are very general. They are not hard to satisfy.
They thus potentially apply to many systems.

Given a quasi-periodic architecture $(T^{\mathrm{nom}}, \varepsilon)$, we can

- Bound the delay between the generation and use of a value,
- Bound overwrites, the number of values lost due to undersampling,
- Bound oversamples, the number of times a single value is read, and,

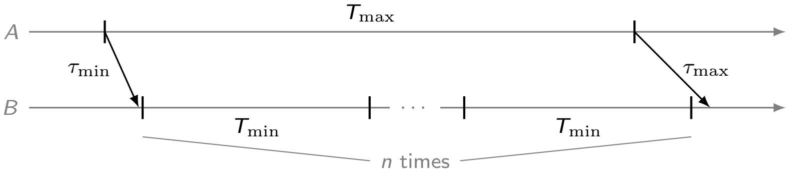
- Derive a sound discrete abstraction: quasi-synchrony.
- Implement synchronous specifications: LTTA.

Quasi-periodic architecture: Sampling effects

Given a pair of nodes executing and communicating in a quasi-periodic architecture, the maximum number of consecutive oversamplings and overwritings is

$$n_{os} = n_{ow} = \left\lceil \frac{T_{\max} + \tau_{\max} - \tau_{\min}}{T_{\min}} \right\rceil - 1. \qquad (1)$$

Oversampling

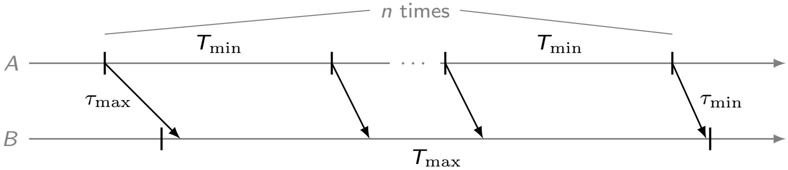


Each execution of B samples the first message sent by A.

Given a pair of nodes executing and communicating in a quasi-periodic architecture, the maximum number of consecutive oversamplings and overwritings is

$$n_{os} = n_{ow} = \left\lceil \frac{T_{\max} + \tau_{\max} - \tau_{\min}}{T_{\min}} \right\rceil - 1. \tag{1}$$

Overwriting



All messages sent by A are overwritten by the last one.

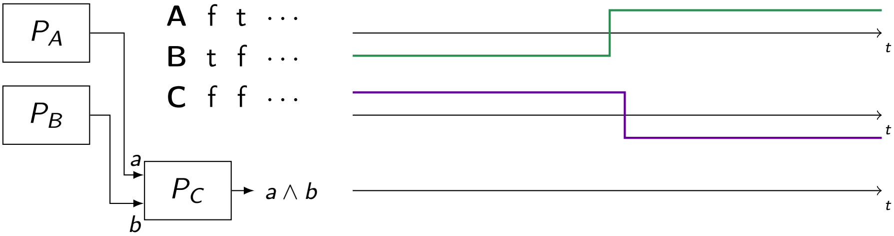
Quasi-periodic architectures: distributed applications

- The quasi-periodic architecture is natural for control applications.
  - The error due to sampling artifacts can be quantified and compensated.
  - Feedback loops are often robust to oversampling and overwriting.

- But what about discrete control logic?
  - Combinations of boolean values are not robust in general.
  - Sequential logic, e.g., state machines, is very sensitive to lost events.
  - How can we guarantee the reference semantics of a program?

Quasi-periodic architectures: distributed applications

- The quasi-periodic architecture is natural for control applications.
  - The error due to sampling artifacts can be quantified and compensated.
  - Feedback loops are often robust to oversampling and overwriting.
- But what about discrete control logic?
  - Combinations of boolean values are not robust in general.
  - Sequential logic, e.g., state machines, is very sensitive to lost events.
  - How can we guarantee the reference semantics of a program?



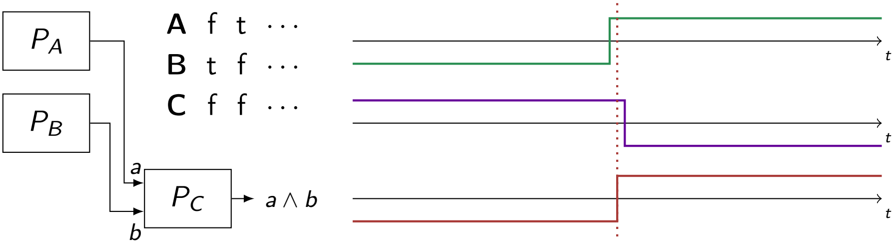
- The quasi-periodic architecture is natural for control applications.
  - The error due to sampling artifacts can be quantified and compensated.
  - Feedback loops are often robust to oversampling and overwriting.
- But what about discrete control logic?
  - Combinations of boolean values are not robust in general.
  - Sequential logic, e.g., state machines, is very sensitive to lost events.
  - How can we guarantee the reference semantics of a program?

Quasi-periodic architectures: distributed applications

- The quasi-periodic architecture is natural for control applications.
  - » The error due to sampling artifacts can be quantified and compensated.
  - » Feedback loops are often robust to oversampling and overwriting.
- But what about discrete control logic?
  - » Combinations of boolean values are not robust in general.
  - » Sequential logic, e.g., state machines, is very sensitive to lost events.
  - » How can we guarantee the reference semantics of a program?

Quasi-periodic architectures: distributed applications

- The quasi-periodic architecture is natural for control applications.
  - The error due to sampling artifacts can be quantified and compensated.
  - Feedback loops are often robust to oversampling and overwriting.
- But what about discrete control logic?
  - Combinations of boolean values are not robust in general.
  - Sequential logic, e.g., state machines, is very sensitive to lost events.
  - How can we guarantee the reference semantics of a program?



- Our idealized parameters abstract from the precise causes of jitter (clock drift, delays in OS, predictability of execution platform, etc.).
- Sampling effects occur as soon as $\varepsilon > 0$.

figures/tintin_comms.pdf

[Hergé (1955): Les Cigares du Pharaon](p.11) figures/tintin_comms.pdf

Message loss?

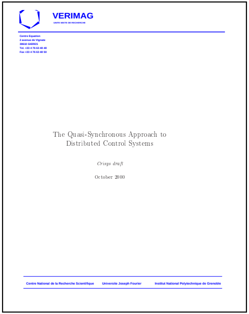
[Hergé (1955): Les Cigares du Pharaon](p.11)

Outline

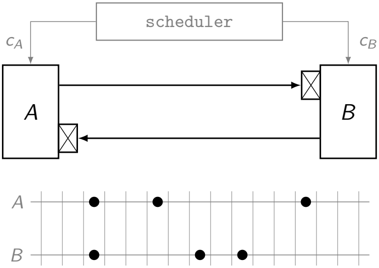
The quasi-synchronous approach

P. Caspi's 'cooking book' [Caspi (2000): The Quasi-Synchronous Approach to Distributed Control Systems]

- A set of techniques for building distributed control systems.
- Presented as a formalization of current practice.
- Such systems are common in aerospace, nuclear power, and rail transportation.



The quasi-synchronous abstraction



Discrete-time model (DT)

Discrete abstraction of the quasi-periodic architecture. Two main ideas:

1. Model (limited) asynchronous interleavings with ‘scheduler’ inputs.
2. Model message transmission as a unit-delay (on model base-clock).

Why? Verify properties of real-time models in the simpler discrete-time model using standard tools.

Modelling asynchrony[5] [Halbwachs and Baghdadi (2002): Synchronous modelling of asynchronous systems]

Standard technique for modelling asynchrony in a synchronous setting:

1. Processes stutter, i.e., do not change state,
   - Interleaving: one process acts, all others *stutter*;
   - Handshake/Rendez-vous: sender and receiver(s) act, all others stutter.
2. Model non-deterministic activation with additional inputs, i.e., clocks in Lustre/Lucid Synchrone, Activation Conditions in SCADE.

[5] R. Milner (1989). *Communication and Concurrency*. SCCS, Chapter 9

Modelling asynchrony[5] [Halbwachs and Baghdadi (2002): Synchronous modelling of asynchronous systems]

Standard technique for modelling asynchrony in a synchronous setting:

1. Processes stutter, i.e., do not change state,
   - Interleaving: one process acts, all others *stutter*;
   - Handshake/Rendez-vous: sender and receiver(s) act, all others stutter.
2. Model non-deterministic activation with additional inputs, i.e., clocks in Lustre/Lucid Synchrone, Activation Conditions in SCADE.

```
node twonodes (c1, c2 : bool, i1, i2 : 'a) returns (mo1, mo2 : 'b);
var o1 : 'b when c1, o2 : 'b when c2, lmo1, lmo2 : 'b;
let
  o1   = node1(i1 when c1);
  mo1  = merge c1 o1 (lmo1 when not c1);
  lmo1 = def1 fby mo1;

  o2   = node2(i2 when c2);
  mo2  = merge c2 o2 (lmo2 when not c2);
  lmo2 = def2 fby mo2;
tel;
```

[5] R. Milner (1989). *Communication and Concurrency*. SCCS, Chapter 9

Restrictions on clocks

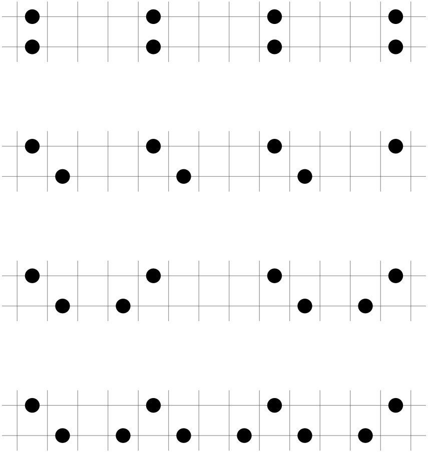
- Allowing arbitrary interleavings is not restrictive enough.
- The timing properties of the architecture (n_o and n_s) effectively preclude certain interleavings:
  Neither of the clocks can take the value 1 more than twice between two successive 1 values of the other.[6] $n_o = n_s = 1$
- More formally, the vector stream composed of the two clocks should never contain the subsequences:

$$\begin{bmatrix}1\\ \_\end{bmatrix} \cdot \begin{bmatrix}0\\0\end{bmatrix}^* \cdot \begin{bmatrix}1\\0\end{bmatrix} \cdot \begin{bmatrix}0\\0\end{bmatrix}^* \cdot \begin{bmatrix}1\\ \_\end{bmatrix} \quad \text{or} \quad \begin{bmatrix}\_\\1\end{bmatrix} \cdot \begin{bmatrix}0\\0\end{bmatrix}^* \cdot \begin{bmatrix}0\\1\end{bmatrix} \cdot \begin{bmatrix}0\\0\end{bmatrix}^* \cdot \begin{bmatrix}\_\\1\end{bmatrix},$$

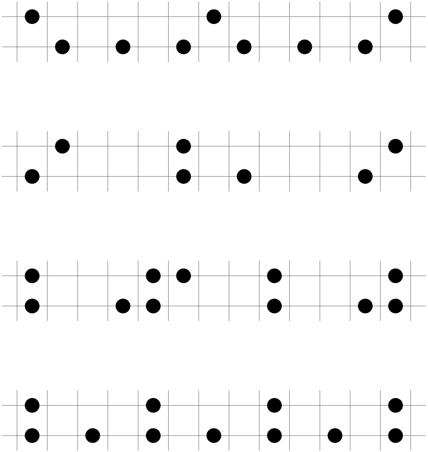
  where _ stands for either 0 or 1.

[6] P. Caspi (May 2000). *The Quasi-Synchronous Approach to Distributed Control Systems*. §3.2.2

Example traces



Quasi-synchronous traces



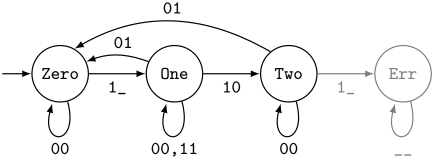
Non-quasi-synchronous traces

$$\begin{bmatrix} \_ \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \end{bmatrix}^* \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \end{bmatrix}^* \cdot \begin{bmatrix} \_ \\ 1 \end{bmatrix}$$

Checking quasi-synchrony

No more than two ticks of c1 between two ticks of c2.

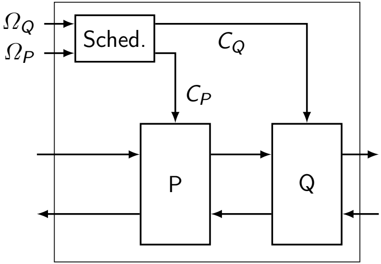
$$\begin{bmatrix}1\\ \_\end{bmatrix} \cdot \begin{bmatrix}0\\0\end{bmatrix}^* \cdot \begin{bmatrix}1\\0\end{bmatrix} \cdot \begin{bmatrix}0\\0\end{bmatrix}^* \cdot \begin{bmatrix}1\\ \_\end{bmatrix}$$

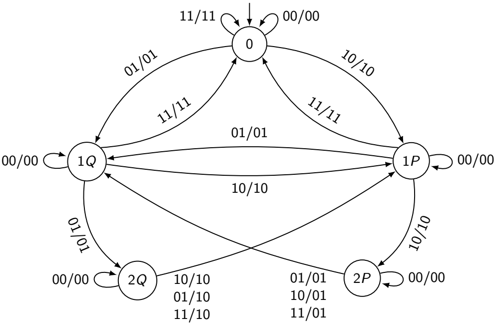


Labels denote clock activations: 01 means, for instance, that c2 ticks alone. The predicate is only *true* in the black region.

```
let node check_qs(c1, c2) = ok where
  rec automaton
      | Zero → do ok = true unless c1() then One
      | One  → do ok = true unless c1() & c2() then One
                              else c1() then Two
                              else c2() then Zero
      | Two  → do ok = true unless c1() then Err
                              else c2() then Zero
      | Err  → do ok = false done

val check_qs: unit signal × unit signal -D-> bool
```

Case study: EADS Space Transportation Proximity Flight Safety System
Automatic Transfer Vehicle supplying International Space Station





Model in Lustre. Verification in Lesar.

Modelling communication with unit delays


```
node twonodes (c1, c2 : bool, i1, i2 : 'a) returns (mo1, mo2 : 'b);
var o1 : 'b when c1, o2 : 'b when c2, lmo1, lmo2 : 'b;
let
  o1   = node1(i1 when c1, lmo2 when c1);
  mo1  = merge c1 o1 (lmo1 when not c1);
  lmo1 = def1 fby mo1;

  o2   = node2(i2 when c2, lmo1 when c2);
  mo2  = merge c2 o2 (lmo2 when not c2);
  lmo2 = def2 fby mo2;
tel;
```

...the delay accounts for short undetermined transmission delays. ...Significantly shorter than the periods of read and write clocks. If longer transmission delays are needed, modeling should be more complex.

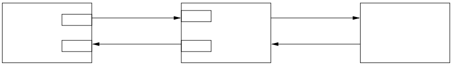
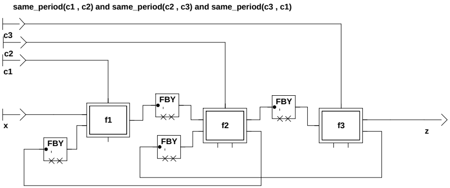


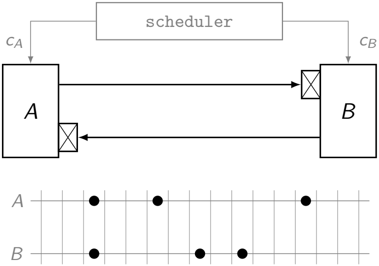
Figure 2.2: A point to point network

Figure 2.2 shows a typical situation borrowed from an automatic subway application. Each computer monitors a rail track section and runs a periodic program. Computers are linked together by serial lines according to the topology of the track. Thus trains passing from a track section to another one are followed by the computers.



We are thus in a position to faithfully represent, simulate, test generate and even formally prove quasi-synchronous programs. For instance, a faithful Scade representation of the architecture shown at figure 2.2 is displayed at figure 3.2.

Verifying quasi-synchronous models



Discrete-time model (DT)

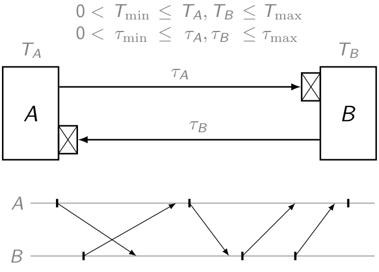
- Use check_qs to constrain clocks using either *assertions* in Lustre/Lesar [Halbwachs, Lagnier, and Ratel (1992): Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE] or *assumptions* in Kind2 [Hagen and Tinelli (2008): Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques].
- Check properties of complete distributed embedded systems with tools usually used for the application alone.

But is the abstraction correct?

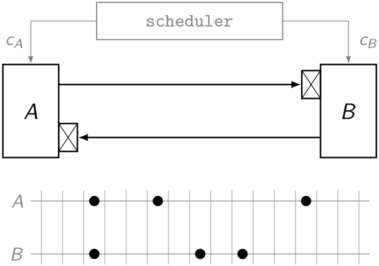
But is the abstraction correct?

What does it mean to be correct?

Correctness of the quasi-synchronous abstraction

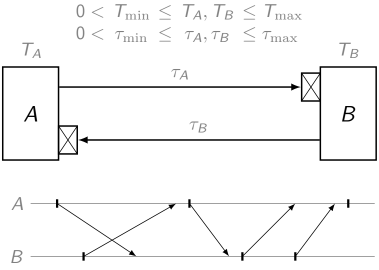


Real-time model (RT)

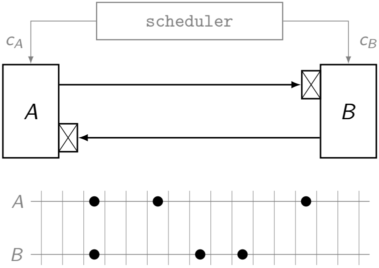


Discrete-time model (DT)

Correctness of the quasi-synchronous abstraction



Real-time model (RT)



Discrete-time model (DT)

- Soundness: A (safety) property φ true for the discrete-time model must also hold for the real-time model: $RT \models \varphi \impliedby DT \models \varphi$.
- All reachable states in the real-time model must also be reachable in the discrete-time model: reachable(RT) $\subseteq$ reachable(DT).

Is the abstraction sound?

- Caspi reasoned that the abstraction was sound for nodes that execute *almost periodically* ($T_{\mathrm{min}} \approx T_{\mathrm{max}}$) when transmission delays are *significantly shorter than the periods of [node activations]*.
- We wanted to make these statements more precise, but instead found that the model is not sound in general for more than two nodes.
- Fortunately, soundness can be recovered under certain conditions...

Formal model for reasoning about soundness

Fix an arbitrary quasi-periodic architecture with nodes $\mathcal{N}$ and parameters $T_{\min}$, $T_{\max}$, $\tau_{\min}$, and $\tau_{\max}$.

Formalize pairs of sending and receiving nodes using a *communicates-with* relation ($\rightrightarrows$) between the nodes.

Definition 2 (Trace)

A *trace* $\mathcal{E} = \{A_i \mid A \in \mathcal{N} \wedge i \in \mathbb{N}\}$ is a set of activation events and two functions:
$t(A_i)$, the date of event A_i with respect to an ideal reference clock,
$\tau(A_i, B)$, the transmission delay of the message sent at A_i to a node B.

These functions satisfy the QPA constraints, namely if $A \rightrightarrows B$,

$$\begin{array}{lcccl} 0 \leq & T_{\min} \leq & t(A_{i+1}) - t(A_i) & \leq T_{\max}, \text{ and} \\ 0 \leq & \tau_{\min} \leq & \tau(A_i, B) & \leq \tau_{\max}. \end{array}$$

(no need to model send or receive events. . .)

Formal model for reasoning about soundness

Definition 2 (Trace)

A *trace* $\mathcal{E} = \{A_i \mid A \in \mathcal{N} \wedge i \in \mathbb{N}\}$ is a set of activation events and two functions:
$t(A_i)$, the date of event A_i with respect to an ideal reference clock,
$\tau(A_i, B)$, the transmission delay of the message sent at A_i to a node B.

These functions satisfy the QPA constraints, namely if $A \rightrightarrows B$,

$$\begin{array}{lclcl} 0 \leq & T_{\min} & \leq t(A_{i+1}) - t(A_i) & \leq T_{\max}, \text{ and} \\ 0 \leq & \tau_{\min} & \leq \tau(A_i, B) & \leq \tau_{\max}. \end{array}$$

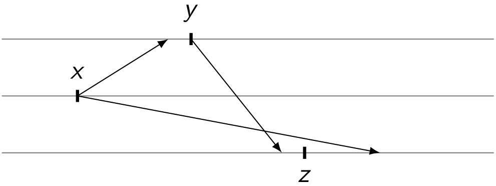
Definition 3 (Happened before)

For a trace $\mathcal{E}$, let $\rightarrow$ be the smallest relation on events that satisfies

(*local*) If $i < j$ then $A_i \rightarrow A_j$, and

(*recv*) If $A \rightrightarrows B$ and $t(A_i) + \tau(A_i, B) \leq t(B_j)$ then $A_i \rightarrow B_j$.

- The $\rightarrow$ relation is a standard [Lamport (1977): Time, Clocks, and the Ordering of Events in a Distributed System] way to model *causality*.
- Except that here it is not closed by transitivity:



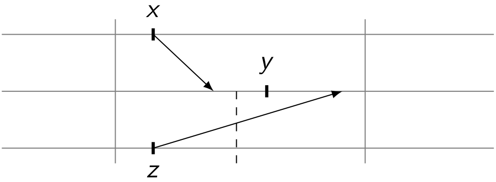
Here $x \not\rightarrow z$, but $x \rightarrow y \rightarrow z$.

- Therefore $x \rightarrow y$ means that y occurs strictly after the reception of the message sent at x.

Discretizing traces

- Discretization gives a total order on (activation) events.
- Transmissions can be rephrased in terms of precedence: if an event x occurs strictly before another event y, the message sent at x is received before y.
- Direct consequence of unit-delay model. Very constraining.

- For example,

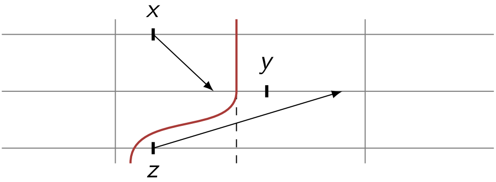


x and y must be in two different instants and z cannot be before y

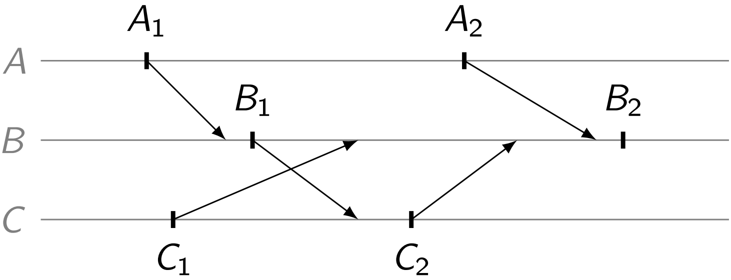
Discretizing traces

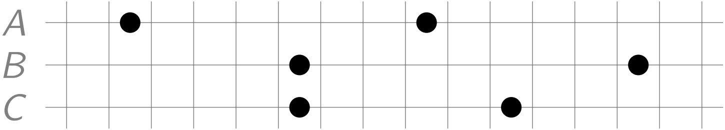
- Discretization gives a total order on (activation) events.
- Transmissions can be rephrased in terms of precedence: if an event x occurs strictly before another event y, the message sent at x is received before y.
- Direct consequence of unit-delay model. Very constraining.

- For example,



  x and y must be in two different instants and z cannot be before y
- To capture transmission we must 'bend' the fences between logical steps: the link between real and discrete time is not based on sampling, but rather on causality.


A_1
A_2
A
B_1
B_2
B
C
C_1
C_2


A
B
C

Definition 4 (Unitary discretization)

A function $f : \mathcal{E} \to \mathbb{N}$ that assigns each event in a (real-time) trace $\mathcal{E}$ to a logical instant of a corresponding discrete trace is a *unitary discretization* if for all $A_i, B_j \in \mathcal{E}$,

$$A_i \to B_j \iff (f(A_i) < f(B_j) \text{ and } A \rightrightarrows B)\,. \tag{UD}$$

- $A_i \to B_j \implies (f(A_i) < f(B_j) \text{ and } A \rightrightarrows B)$
  since the $\to$ relation induces a partial order on events.

- $(f(A_i) < f(B_j) \text{ and } A \rightrightarrows B) \implies A_i \to B_j$
  for communicating nodes $A \rightrightarrows B$, if an event B_j occurs after an event A_i in the discrete-time model, that is, $f(A_i) < f(B_j)$, either
  - » B_j is a later activation of the same node as A_i ($A = B$ and $j > i$), or
  - » B_j occurs strictly after the receipt of the message sent at A_i.

- A unitary discretization links the causality of events in the real-time model to the causality implicit in the discrete-time model.

Unitary Discretization

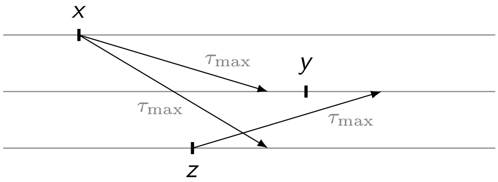
Definition 4 (Unitary discretization)

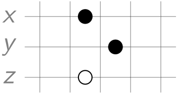
A function $f : \mathcal{E} \to \mathbb{N}$ that assigns each event in a (real-time) trace $\mathcal{E}$ to a logical instant of a corresponding discrete trace is a *unitary discretization* if for all $A_i, B_j \in \mathcal{E}$,

$$A_i \to B_j \iff (f(A_i) < f(B_j) \text{ and } A \rightrightarrows B)\,. \tag{UD}$$

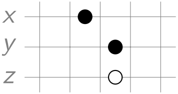
Definition 5 (Unitary discretizable)

A quasi-periodic architecture is *unitary discretizable* if for each of its possible traces there exists a unitary discretization.





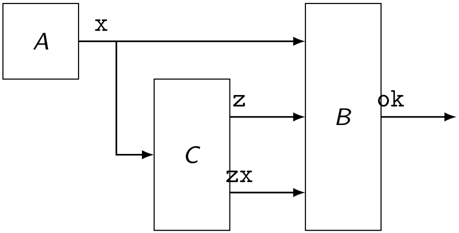
$z \to y$

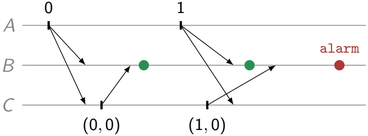


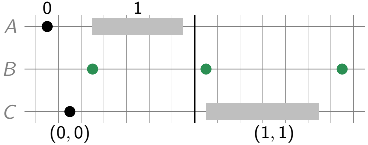
$x \to z$

- There are real-time traces that are not unitary discretizable.
- Possible for 3 inter-communicating nodes whenever $\tau_{\max} > 0$.
- Does it matter?

```
let node a() = x where
  rec x = 0 fby (x + 1)

let node c(x) = z, zx where
  rec z = 0 fby (z+1)
  and zx = x

let node b(x, z, zx) = ok where
  rec cx = (x = (0 fby x))
  and cz = (z > (0 fby z))
  and czx = (x > zx)
  and ok = not (cx && cz && czx)
```







- cx: no new message from A at B (since last activation of B).
- cz: new message from C at B.
- czx: x from A fresher than x via C.

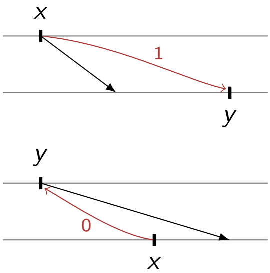
- Can happen in real-time trace.
- Impossible in discrete-time trace.

Trace Graphs

In a unitary discretization f of a trace, for a pair of nodes $A \rightrightarrows B$:

$$A_i \to B_j \implies f(A_i) < f(B_j), \text{ and}$$
$$A_i \not\to B_j \implies f(A_i) \geq f(B_j).$$

capture constraints in a graph:



Definition 6 (Trace graph)

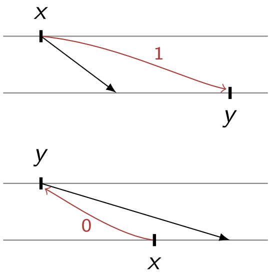
Given a trace $\mathcal{E}$, its directed, weighted *trace graph* $\mathcal{G}$ has as vertices $\{A_i \mid A \in \mathcal{N} \wedge i \in \mathbb{N}\}$ and as edges the smallest relations that satisfy

1. If $A_i \to B_j$ then $A_i \xrightarrow{1} B_j$.
2. If $A \rightrightarrows B$ and $A_i \not\to B_j$ then $B_j \xrightarrow{0} A_i$.

In a unitary discretization f of a trace, for a pair of nodes $A \rightrightarrows B$:

$$A_i \to B_j \implies f(A_i) < f(B_j), \text{ and}$$
$$A_i \not\to B_j \implies f(A_i) \geq f(B_j).$$

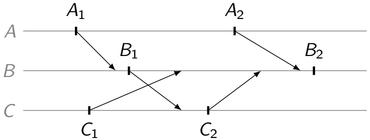
capture constraints in a graph:

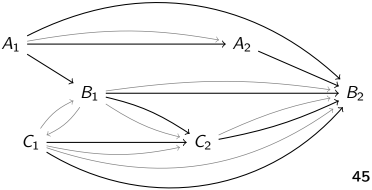


Definition 6 (Trace graph)

Given a trace $\mathcal{E}$, its directed, weighted *trace graph* $\mathcal{G}$ has as vertices $\{A_i \mid A \in \mathcal{N} \wedge i \in \mathbb{N}\}$ and as edges the smallest relations that satisfy

1. If $A_i \to B_j$ then $A_i \xrightarrow{1} B_j$.
2. If $A \rightrightarrows B$ and $A_i \not\to B_j$ then $B_j \xrightarrow{0} A_i$.



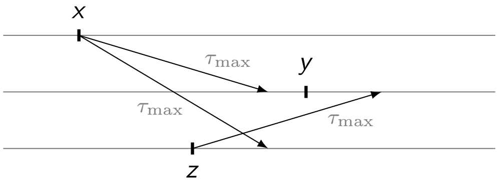


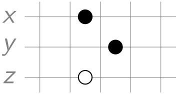
- $x \xrightarrow{1} y$ represents $f(x) < f(y)$
  Source activation always before destination activation
  (f must strictly increase in any unitary discretization).
- $x \xrightarrow{0} y$ represents $f(x) \leq f(y)$
  Source activation never before destination activation
  (f must be the same or larger in any unitary discretization).
- A path through several activations defines their relative ordering in all possible unitary discretizations.
- Constraint satisfaction now expressible in terms of cycles:
  - Cycle of $\xrightarrow{0}$ is OK: all in same slot.
  - Cycle with $\xrightarrow{1}$ is KO: contradictory constraints.

- $x \xrightarrow{1} y$ represents $f(x) < f(y)$
  Source activation always before destination activation
  (f must strictly increase in any unitary discretization).
- $x \xrightarrow{0} y$ represents $f(x) \leq f(y)$
  Source activation never before destination activation
  (f must be the same or larger in any unitary discretization).
- A path through several activations defines their relative ordering in all possible unitary discretizations.
- Constraint satisfaction now expressible in terms of cycles:
  - Cycle of $\xrightarrow{0}$ is OK: all in same slot.
  - Cycle with $\xrightarrow{1}$ is KO: contradictory constraints.

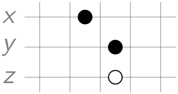
Lemma 7 ($\exists$UD $\iff$ $\overline{\exists \text{PC}}$)

For a trace $\mathcal{E}$, there is a unitary discretization ($\exists$UD) *if and only if there is no cycle of positive weight in the corresponding trace graph $\mathcal{G}$* ($\overline{\exists \text{PC}}$)*.*

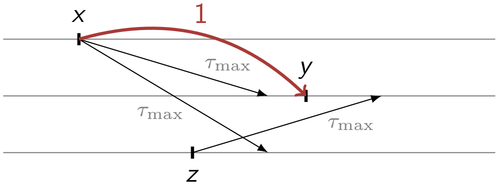


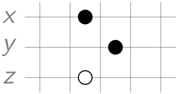


$z \to y$

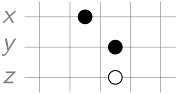


$x \to z$

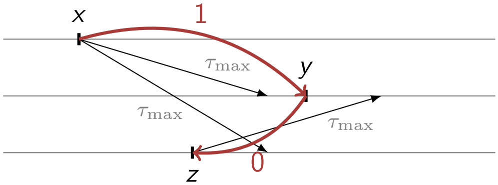


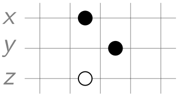


$z \to y$

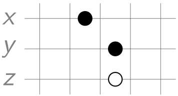


$x \to z$

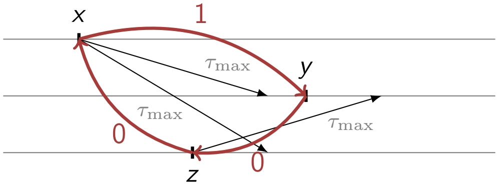


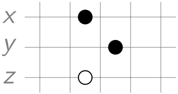


$z \to y$

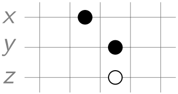


$x \to z$





$z \to y$



$x \to z$

Recovering Soundness

Prevent problematic traces by:

- Constraining the timing parameters $T_{\min}$, $T_{\max}$, $\tau_{\min}$, and $\tau_{\max}$.
- Restricting the communication graph:
  forbidding $A \rightrightarrows B$ removes $A_i \xrightarrow{1} B_j$ and $B_j \xrightarrow{0} A_i$, for all i and j, in associated trace graphs.

$$\begin{tikzcd}
A \arrow[r, shift left] \arrow[r, shift right] & B \arrow[d, shift left] \arrow[d, shift right] \\
D \arrow[u, shift left] \arrow[u, shift right] & C \arrow[l, shift left] \arrow[l, shift right]
\end{tikzcd}$$

Cycle

$$\begin{tikzcd}
A \arrow[rr, shift left] \arrow[rr, shift right] \arrow[dr, shift left] \arrow[dr, shift right] & & B \\
& C \arrow[ur, shift left] \arrow[ur, shift right] &
\end{tikzcd}$$

u-cycle

$$\begin{tikzcd}
A \arrow[r, shift left] \arrow[r, shift right] \arrow[d, shift left] \arrow[d, shift right] & B \arrow[d, shift left] \arrow[d, shift right] \\
D \arrow[r, shift left] \arrow[r, shift right] & C
\end{tikzcd}$$

b-cycle

$$\begin{tikzcd}
A & B \arrow[l, shift left] \arrow[l, shift right] \arrow[d, shift left] \arrow[d, shift right] \\
D \arrow[u, shift left] \arrow[u, shift right] \arrow[r, shift left] \arrow[r, shift right] & C
\end{tikzcd}$$

b-cycle

- *u*-cycle: undirected cycle (i.e., ignore edge directions)
- *b*-cycle: balanced *u*-cycle (same number of edges in each direction)

$$\begin{tikzcd} A \arrow[r, shift left] \arrow[r, shift right] & B \arrow[d, shift left] \arrow[d, shift right] \\ D \arrow[u, shift left] \arrow[u, shift right] & C \arrow[l, shift left] \arrow[l, shift right] \end{tikzcd}$$

Cycle

$$\begin{tikzcd} A \arrow[rr, shift left] \arrow[rr, shift right] \arrow[dr, shift left] \arrow[dr, shift right] & & B \\ & C \arrow[ur, shift left] \arrow[ur, shift right] & \end{tikzcd}$$

u-cycle

$$\begin{tikzcd} A \arrow[r, shift left] \arrow[r, shift right] \arrow[d, shift left] \arrow[d, shift right] & B \arrow[d, shift left] \arrow[d, shift right] \\ D \arrow[r, shift left] \arrow[r, shift right] & C \end{tikzcd}$$

b-cycle

$$\begin{tikzcd} A & B \arrow[l, shift left] \arrow[l, shift right] \arrow[d, shift left] \arrow[d, shift right] \\ D \arrow[u, shift left] \arrow[u, shift right] \arrow[r, shift left] \arrow[r, shift right] & C \end{tikzcd}$$

b-cycle

Theorem 8

Let L_c be the size of the longest elementary cycle in the communication graph. A quasi-periodic architecture is unitary discretizable if and only if, the three following conditions hold:

1. *All u-cycles of the communication graph are cycles, or balanced u-cycles, or $\tau_{\max} = 0$.*
2. *There is no balanced u-cycle in the communication graph or $\tau_{\min} = \tau_{\max}$.*
3. *There is no cycle in the communication graph, or*

$$T_{\min} \geq L_c \tau_{\max}. \qquad \text{(CD)}$$

Cycle of positive weight from cycle in comm. graph

$$\begin{tikzcd}
A \arrow[d, shift right] \arrow[d, shift left] & B \arrow[l, shift right] \arrow[l, shift left] \\
D \arrow[r, shift right] \arrow[r, shift left] & C \arrow[u, shift right] \arrow[u, shift left]
\end{tikzcd}$$

Cycle

e'_0

Cycle of positive weight from cycle in comm. graph

$$\begin{tikzcd} A \arrow[d, shift right] \arrow[d, shift left] & B \arrow[l, shift left] \arrow[l, shift right] \\ D \arrow[r, shift left] \arrow[r, shift right] & C \arrow[u, shift left] \arrow[u, shift right] \end{tikzcd}$$

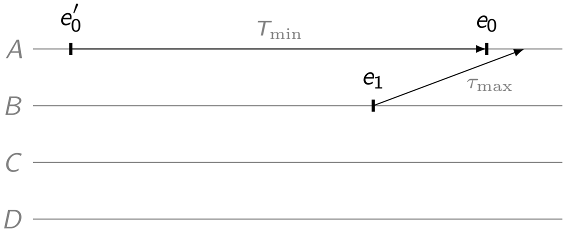
Cycle

$$e_0' \xrightarrow{1} e_0$$

Cycle of positive weight from cycle in comm. graph

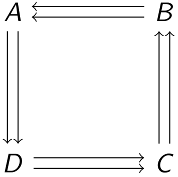
$$\begin{tikzcd} A \arrow[d, shift right] \arrow[d, shift left] & B \arrow[l, shift right] \arrow[l, shift left] \\ D \arrow[r, shift right] \arrow[r, shift left] & C \arrow[u, shift right] \arrow[u, shift left] \end{tikzcd}$$

Cycle

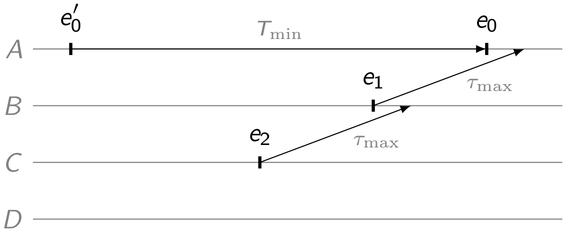


$$e_0' \xrightarrow{1} e_0 \xrightarrow{0} e_1$$

Cycle of positive weight from cycle in comm. graph



Cycle

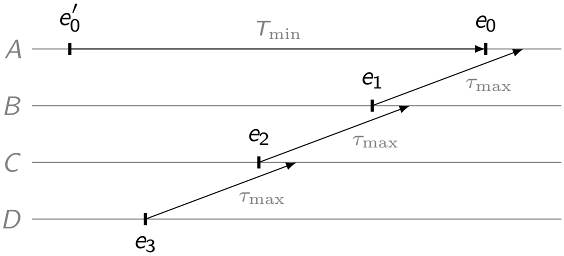


$$e_0' \xrightarrow{1} e_0 \xrightarrow{0} e_1 \xrightarrow{0} e_2$$

Cycle of positive weight from cycle in comm. graph

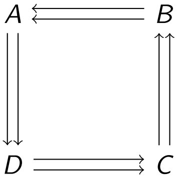
$$\begin{tikzcd} A \arrow[d, shift right, ""] \arrow[d, shift left] & B \arrow[l, shift right] \arrow[l, shift left] \\ D \arrow[r, shift right] \arrow[r, shift left] & C \arrow[u, shift right] \arrow[u, shift left] \end{tikzcd}$$

Cycle

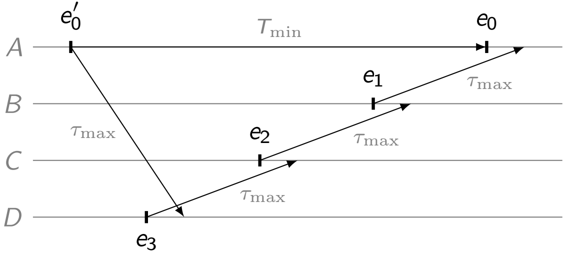


$$e_0' \xrightarrow{1} e_0 \xrightarrow{0} e_1 \xrightarrow{0} e_2 \xrightarrow{0} e_3$$

Cycle of positive weight from cycle in comm. graph

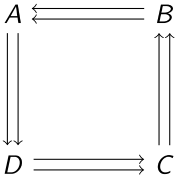


Cycle

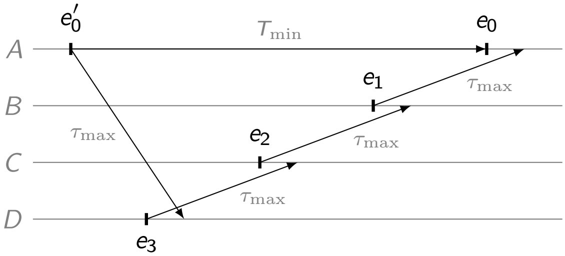


$$e_0' \xrightarrow{1} e_0 \xrightarrow{0} e_1 \xrightarrow{0} e_2 \xrightarrow{0} e_3 \xrightarrow{0} e_0'$$

Cycle of positive weight from cycle in comm. graph



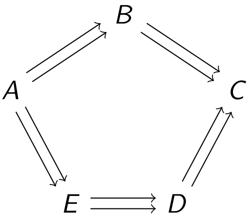
Cycle



$$e_0' \xrightarrow{1} e_0 \xrightarrow{0} e_1 \xrightarrow{0} e_2 \xrightarrow{0} e_3 \xrightarrow{0} e_0'$$

If there is a cycle in the communication graph then require $T_{\min} \geq L_c \tau_{\max}$.

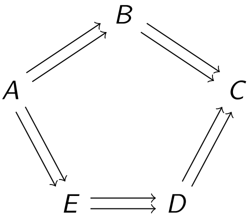
Cycle of positive weight from u-cycle in comm. graph



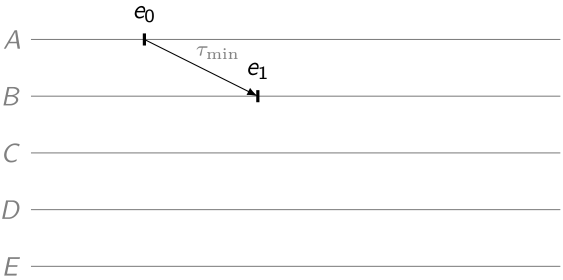
u-cycle

e_0

Cycle of positive weight from u-cycle in comm. graph

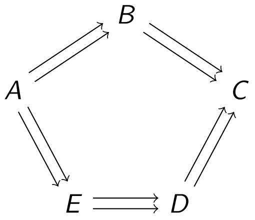


u-cycle

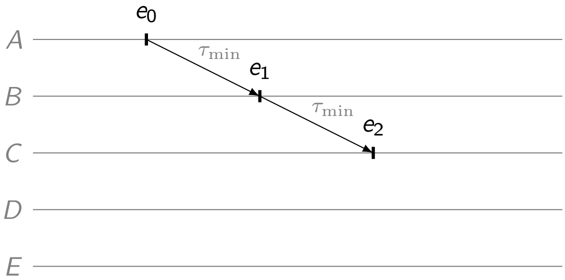


$e_0 \xrightarrow{1} e_1$

Cycle of positive weight from u-cycle in comm. graph

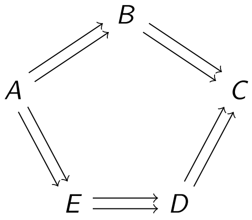


u-cycle

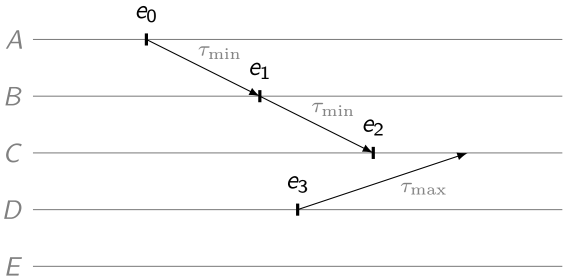


$$e_0 \xrightarrow{1} e_1 \xrightarrow{1} e_2$$

Cycle of positive weight from u-cycle in comm. graph

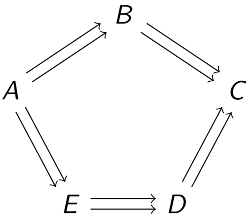


u-cycle

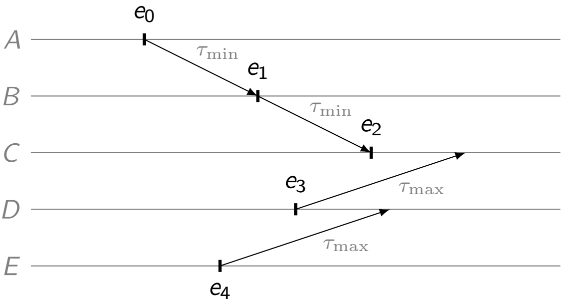


$$e_0 \xrightarrow{1} e_1 \xrightarrow{1} e_2 \xrightarrow{0} e_3$$

Cycle of positive weight from u-cycle in comm. graph

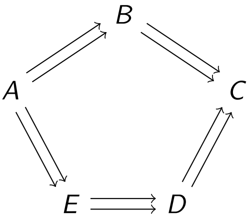


u-cycle

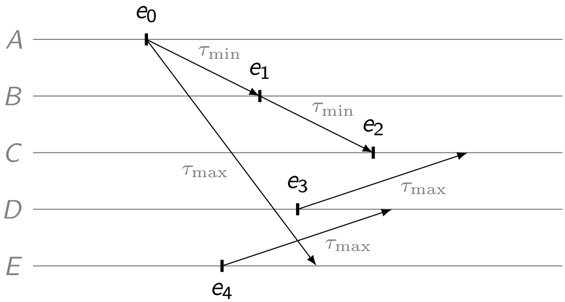


$$e_0 \xrightarrow{1} e_1 \xrightarrow{1} e_2 \xrightarrow{0} e_3 \xrightarrow{0} e_4$$

Cycle of positive weight from u-cycle in comm. graph

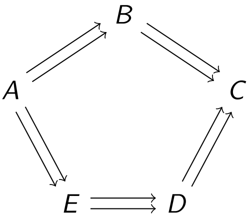


u-cycle

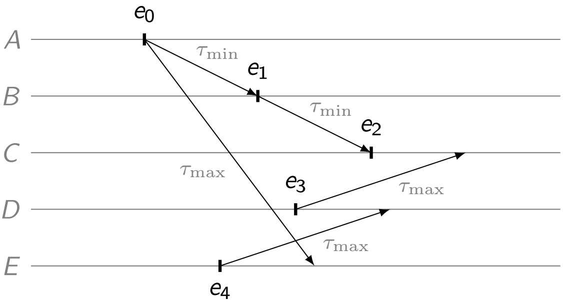


$$e_0 \xrightarrow{1} e_1 \xrightarrow{1} e_2 \xrightarrow{0} e_3 \xrightarrow{0} e_4 \xrightarrow{0} e_0$$

Cycle of positive weight from u-cycle in comm. graph



u-cycle



$$e_0 \xrightarrow{1} e_1 \xrightarrow{1} e_2 \xrightarrow{0} e_3 \xrightarrow{0} e_4 \xrightarrow{0} e_0$$

For a u-cycle that is not a cycle or a b-cycle then $\tau_{\max} = 0$.

Corollary 9 (2-nodes unitary discretization)

A real-time model with two nodes can be unitary discretized if and only if

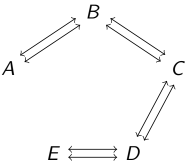
$$T_{\min} \geq 2\tau_{\max}. \tag{2D}$$

Unitary-discretizable Systems

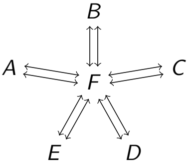
Corollary 9 (2-nodes unitary discretization)

A real-time model with two nodes can be unitary discretized if and only if

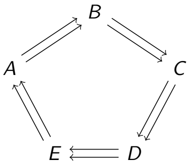
$$T_{\min} \geq 2\tau_{\max}. \tag{2D}$$



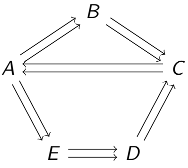
Line: $T_{\min} \geq 2\tau_{\max}$



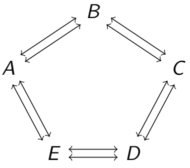
Star: $T_{\min} \geq 2\tau_{\max}$



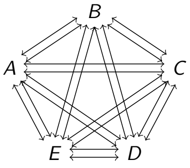
Ring: $T_{\min} \geq 5\tau_{\max}$



Mesh: $\tau_{\max} = 0$



Double ring: $\tau_{\max} = 0$



Clique: $\tau_{\max} = 0$

What about the previous examples?

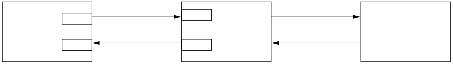
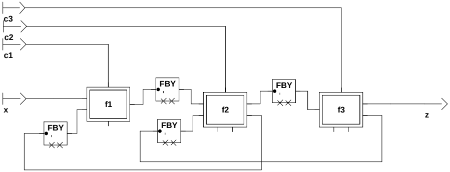


Figure 2.2: A point to point network

same_period(c1 , c2) and same_period(c2 , c3) and same_period(c3 , c1)



What about the previous examples?

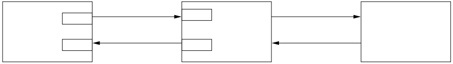
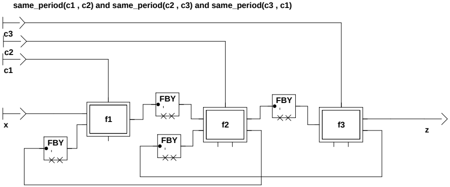


Figure 2.2: A point to point network



OK (line) if $T_{\min} \geq 2\tau_{\max}$

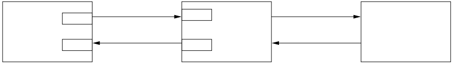
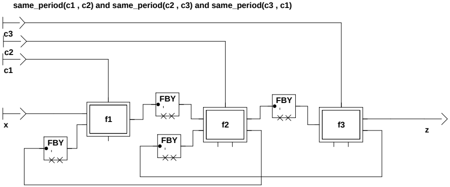
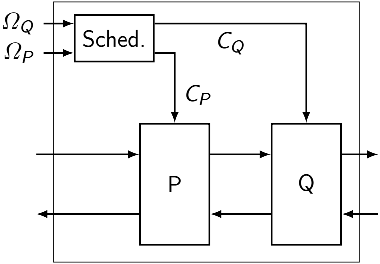


Figure 2.2: A point to point network



OK (line) if $T_{\mathrm{min}} \geq 2\tau_{\mathrm{max}}$



What about the previous examples?

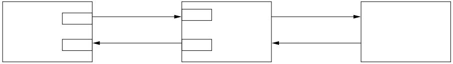
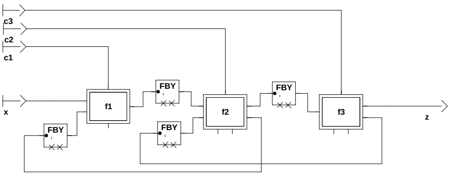
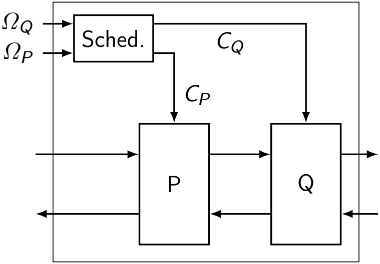


Figure 2.2: A point to point network



OK (line) if $T_{\min} \geq 2\tau_{\max}$



OK (2 nodes) if $T_{\min} \geq 2\tau_{\max}$

Redefining quasi-synchrony

Recall [Caspi (2001): Embedded Control: From Asynchrony to Synchrony and Back]: a discrete-time model is termed quasi-synchronous if

> *It is not the case that a component process executes more than twice between two successive executions of another process.*

Since a node only detects the activations of another by receiving its messages, the quasi-synchronous condition corresponds to two constraints.

Definition 10 (Quasi-Synchronous Model)

A real-time model is quasi-synchronous if, for every trace $\mathcal{E}$,

1. it has a unitary discretization f, and
2. for nodes $A \leftleftarrows B$, there are no i and j such that

$$\begin{aligned} & f(B_j) < f(A_i) < f(A_{i+2}) \leq f(B_{j+1}) \text{ or,} \\ & f(A_j) \leq f(B_i) < f(B_{i+2}) < f(A_{j+1}). \end{aligned} \tag{QS}$$

Redefining quasi-synchrony

Recall [Caspi (2001): Embedded Control: From Asynchrony to Synchrony and Back]: a discrete-time model is termed quasi-synchronous if

> *It is not the case that a component process executes more than twice between two successive executions of another process.*

Since a node only detects the activations of another by receiving its messages, the quasi-synchronous condition corresponds to two constraints.

Definition 10 (Quasi-Synchronous Model)

A real-time model is quasi-synchronous if, for every trace $\mathcal{E}$,

1. it has a unitary discretization f, and
2. for nodes $A \leftleftarrows B$, there are no i and j such that

$$\begin{aligned} &f(B_j) < f(A_i) < f(A_{i+2}) \leq f(B_{j+1}) \text{ or,} \\ &f(A_j) \leq f(B_i) < f(B_{i+2}) < f(A_{j+1}). \end{aligned} \tag{QS}$$

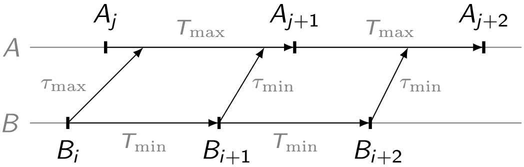
Expresses the two aspects of quasi-synchrony: communications as logical unit delays, and constraints on interleavings of node activations.

Too restrictive?

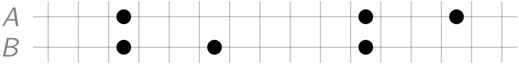
$$\neg\left(\begin{bmatrix}1\\\_\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}1\\0\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}1\\\_\end{bmatrix}\quad\vee\quad\begin{bmatrix}\_\\1\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}0\\1\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}\_\\1\end{bmatrix}\right)$$

$$\neg\left(\begin{bmatrix}1\\\_\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}1\\0\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}1\\\_\end{bmatrix}\quad\vee\quad\begin{bmatrix}\_\\1\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}0\\1\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}\_\\1\end{bmatrix}\right)$$

Take $T_{\min} + \tau_{\min} < T_{\max} + \tau_{\max}$ ($A \leftleftarrows B$ but $A \not\rightleftarrows B$):



Valid unitary discretization:

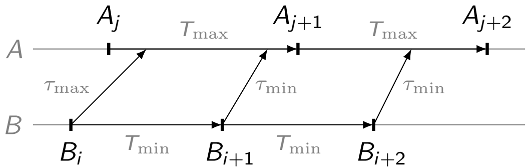


Too restrictive?

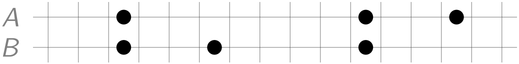
$$\neg\left(\begin{bmatrix}1\\ \underline{0}\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}1\\0\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}1\\ \_\end{bmatrix} \quad\vee\quad \begin{bmatrix}\_\\1\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}0\\1\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}\underline{0}\\1\end{bmatrix}\right)$$

$$f(B_j) < f(A_i) < f(A_{i+2}) \leq f(B_{j+1}) \quad\vee\quad f(A_j) \leq f(B_i) < f(B_{i+2}) < f(A_{j+1})$$

Take $T_{\mathrm{min}} + \tau_{\mathrm{min}} < T_{\mathrm{max}} + \tau_{\mathrm{max}}$ ($A \Leftarrow B$ but $A \not\Rightarrow B$):



Valid unitary discretization:



Redefining quasi-synchrony 2

Definition 11 (Quasi-Synchronous Model)

A real-time model is quasi-synchronous if, for every trace $\mathcal{E}$,

1. it has a unitary discretization f, and
2. for nodes $A \leftleftarrows B$, there are no i and j such that

$$\begin{aligned} &f(B_j) < f(A_i) < f(A_{i+2}) \leq f(B_{j+1}) \text{ or,} \\ &f(A_j) \leq f(B_i) < f(B_{i+2}) < f(A_{j+1}). \end{aligned} \tag{QS}$$

Theorem 12

A quasi-synchronous architecture is quasi-synchronous if and only if,

1. *the previous conditions on communication graphs and timing parameters hold, and*
2. *the following condition holds,*

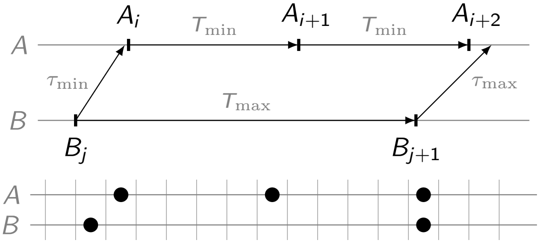
$$2T_{\min} + \tau_{\min} \geq T_{\max} + \tau_{\max}. \tag{QT}$$

Theorem 12

A quasi-synchronous architecture is quasi-synchronous if and only if,

1. *the previous conditions on communication graphs and timing parameters hold, and*
2. *the following condition holds,*

$$2T_{\min} + \tau_{\min} \geq T_{\max} + \tau_{\max}. \tag{QT}$$



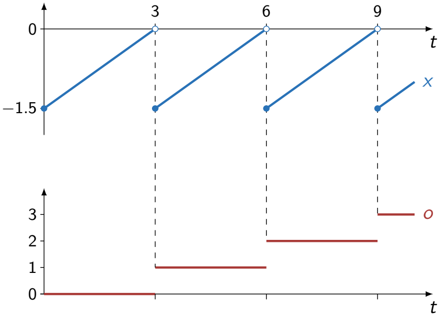
Quasi-synchrony: summary

- The quasi-synchronous model and its limitations is now well understood. [Baudart, Bourke, and Pouzet (2016): Soundness of the Quasi-Synchronous Abstraction]
  - Is it possible to characterize a class of programs that never exploits the 'causality gap' between RT and DT?
  - Is is possible to characterize a class of properties that cannot detect the 'causality gap'?
- The problem is not the treatment of interleavings; the modelling of communications by a unit delay is insufficient in general.
- The model can sometimes be used for verification but not always. Perhaps surprising that it can be used at all?
- Natural generalization to m/n-quasi-synchrony.

Outline

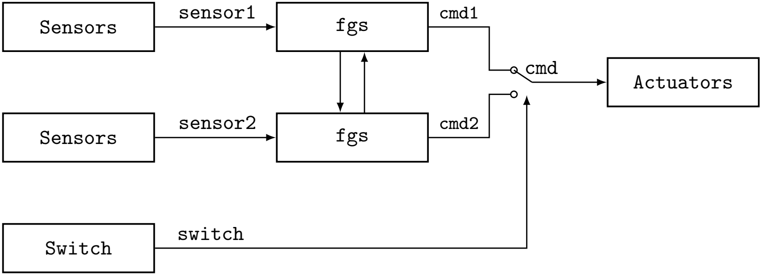
Zélus: synchronous language + ODEs [Bourke and Pouzet (2013): Zélus: A Synchronous Language with ODEs]

```
let node nat(v) = y where
  rec y = v fby (y + 1)

let hybrid sawtooth(x', x0) = o where
  rec init o = 0
  and der x = x' init x0 reset z → x0
  and z = up(x)
  and present z → do o = nat(1) done

let hybrid main = sawtooth(0.5, −1.5)
```



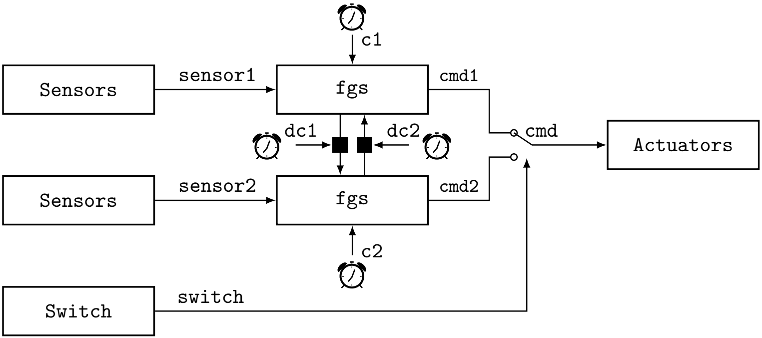
- Combine discrete-time and continuous-time behaviours
  - A type system ensures that compositions are well-defined.
  - Align discrete behaviours on 'zero-crossing' events.
- Source-to-source compilation for simulation with a numeric solver.
- Research focus on hybrid programming languages
  - E.g., Simulink/Stateflow, Modelica, Ptolemy...
- Manual and compiler: http://zelus.di.ens.fr

Return to the Flight Guidance System example




```
let node controller(sensor1, sensor2, switch) = cmd where
  rec cmd1 = fgs(sensor1, cmd2)
  and cmd2 = fgs(sensor2, cmd1)
  and cmd = if switch then cmd1 else cmd2
```

Return to the Flight Guidance System example

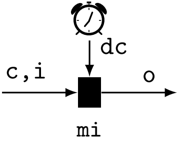


let node qp_controller((c1, c2, dc1, dc2), (sensor1, sensor2, switch)) = cmd where
  rec present c1() → do emit cmd1 = fgs(sensor1, mcmd2) done
  and present c2() → do emit cmd2 = fgs(sensor2, mcmd1) done
  and mcmd1 = link(c1, dc1, cmd1, idle)
  and mcmd2 = link(c2, dc2, cmd2, idle)
  and cmd = if switch then cmd1 else cmd2

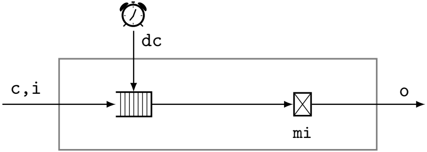
No soundness issues: distinct **activation** and **receive** events.

Modelling links

let node link(c, dc, i, mi) = o where
  rec s = channel(c, dc, i)
  and o = mem(s, mi)

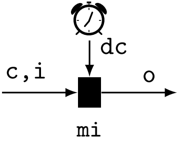


`val link: unit signal` $\times$ `unit signal` $\times\ \alpha$ `signal` $\times\ \alpha \xrightarrow{\mathtt{D}} \alpha$

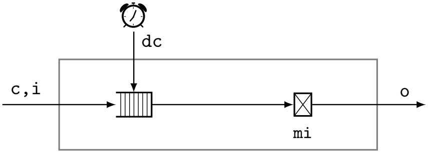


Modelling links

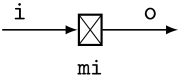
```
let node link(c, dc, i, mi) = o where
  rec s = channel(c, dc, i)
  and o = mem(s, mi)
```



$\texttt{val link: unit signal} \times \texttt{unit signal} \times \alpha\ \texttt{signal} \times \alpha \xrightarrow{\texttt{D}} \alpha$




```
let node mem(i, mi) = o where
  rec init m = mi
  and present i(v) → do m = v done
  and o = last m
```

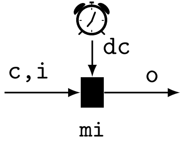


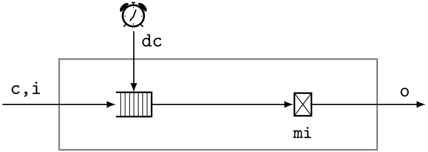
$\texttt{val mem: } \alpha\ \texttt{signal} \times \alpha \xrightarrow{\texttt{D}} \alpha$

Modelling links

```
let node link(c, dc, i, mi) = o where
  rec s = channel(c, dc, i)
  and o = mem(s, mi)
```

val link: unit signal $\times$ unit signal $\times\ \alpha$ signal $\times\ \alpha \xrightarrow{D} \alpha$





- *dc* is a delayed version of *i*.
- It models the transmission delay $(\tau_{\min}, \tau_{\max})$.

```
let node channel(dc, i) = o where
  rec init q = empty
  and present
      | dc() & i(v) → do q = enqueue(dequeue(last q), v) done
      | i(v) → do q = enqueue (last q, v) done
      | dc() → do q = dequeue (last q) done
  and present dc() → do emit o = front(last q) done
```

val channel: unit signal $\times\ \alpha$ signal $\xrightarrow{D} \alpha$ signal

Modelling Real-Time

```
let hybrid rt_controller(sensor1, sensor2, switch) = cmd where
  rec c1 = metro(t_min, t_max)
  and dc1 = delay(c1, tau_min, tau_max)
  and c2 = metro(t_min, t_max)
  and dc2 = delay(c2, tau_min, tau_max)
  and present c1() | dc1() | c2() | dc2() → do emit g done
  and present g() → do cmd = qp_controller ((c1, c2, dc1, dc2),
                                            (sensor1, sensor2, switch)) done
```

$\texttt{val rt\_controller: data} \times \texttt{data} \times \texttt{bool} \xrightarrow{\texttt{C}} \texttt{cmd signal}$

```
let hybrid rt_controller(sensor1, sensor2, switch) = cmd where
  rec c1 = metro(t_min, t_max)
  and dc1 = delay(c1, tau_min, tau_max)
  and c2 = metro(t_min, t_max)
  and dc2 = delay(c2, tau_min, tau_max)
  and present c1() | dc1() | c2() | dc2() → do emit g done
  and present g() → do cmd = qp_controller ((c1, c2, dc1, dc2),
                                            (sensor1, sensor2, switch)) done
val rt_controller: data × data × bool -C-> cmd signal

let hybrid metro(t_min, t_max) = c where
  rec der t = 1.0 init −. arbitrary (t_min, t_max)
      reset z → −. arbitrary (t_min, t_max)
  and z = up(t)
  and present (init) | z → do emit c done
val metro: float × float -C-> unit signal
```

Modelling Real-Time

```
let hybrid rt_controller(sensor1, sensor2, switch) = cmd where
  rec c1 = metro(t_min, t_max)
  and dc1 = delay(c1, tau_min, tau_max)
  and c2 = metro(t_min, t_max)
  and dc2 = delay(c2, tau_min, tau_max)
  and present c1() | dc1() | c2() | dc2() → do emit g done
  and present g() → do cmd = qp_controller ((c1, c2, dc1, dc2),
                                  (sensor1, sensor2, switch)) done
val rt_controller: data × data × bool -C-> cmd signal
```

```
let hybrid delay(c, tau_min, tau_max) = dc where
  rec der t = 1.0 init 0.0 reset c() → −. arbitrary (tau_min, tau_max)
  and present up(t) → do emit dc done
val delay: unit signal × float × float -C-> unit signal
```

- Model transmission delay relative to node activation.
- Captures the intuition, but too simple (mandates $\tau_{\min} < T_{\min}$).
- Need a queue to model simultaneous ongoing transmissions.

Modelling Real-Time

```
let hybrid rt_controller(sensor1, sensor2, switch) = cmd where
  rec c1 = metro(t_min, t_max)
  and dc1 = delay(c1, tau_min, tau_max)
  and c2 = metro(t_min, t_max)
  and dc2 = delay(c2, tau_min, tau_max)
  and present c1() | dc1() | c2() | dc2() → do emit g done
  and present g() → do cmd = qp_controller ((c1, c2, dc1, dc2),
                                            (sensor1, sensor2, switch)) done
val rt_controller: data × data × bool -C-> cmd signal
```

How to express 'arbitrary'?

- For random simulations:

  `let arbitrary(l, u) = l +. Random.float (u −. l)`

- Not very satisfying. Precise semantics is unclear.
- Model nondeterminism with new program inputs?
- ODEs ($\dot{x} = 1$) are overkill: special treatment for timers?

Outline

Implementing Discrete Logic on Quasi-Periodic Systems

- Sampled controllers—e.g., PIDs—are usually robust to the sampling effects described earlier; control theory gives mathematical tools for taking them into account.
- Discrete logic, however, is sensitive to such effects.
- How can we program discrete control logic in a synchronous language and then reliably distribute it across several controllers?
- Typical solution: clock synchronization.
  - [Kopetz and Bauer (2003): The Time-Triggered Architecture]'s Time-Triggered Architecture (TTA)
  - FlexRay protocol, TTEthernet
  - Network Time Protocol (NTP), True-Time (TT)
- Alternative: Loosely Time-Triggered Architecture (LTTA) [Benveniste, Caspi, Le Guernic, Marchand, Talpin, and Tripakis (2002): A Protocol for Loosely Time-Triggered Architectures] protocols:
  - Back-Pressure LTTA
  - Time-Based LTTA
  - Round-Based LTTA

Distributing Synchronous Applications

Compile from Lustre/SCADE, Signal, Esterel, or Simulink into communicating Mealy machines.

A Mealy machine m is a tuple $\langle s_{\mathrm{init}}, I, O, F \rangle$, where

- s_{init} is an initial state,
- I is a set of input variables,
- O is a set of output variables, and
- F is a transition function mapping a state and input values to the next state and output values: $F : \mathcal{S} \times \mathcal{V}^I \to \mathcal{S} \times \mathcal{V}^O$.

The semantics is a stream function[7]

$$[\![m]\!] : (\mathcal{V}^I)^\infty \to (\mathcal{V}^O)^\infty$$

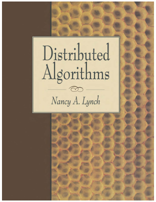
generated by iterating the transition function from the initial state:

$$\begin{aligned} s(0) &= s_{\mathrm{init}} \\ s(n+1),\ o(n) &= F(s(n), i(n)). \end{aligned}$$

[7] $\mathcal{X}^\infty = \mathcal{X}^* \cup \mathcal{X}^\omega$ denotes the set of possibly finite streams over elements in $\mathcal{X}$.

Distributing Synchronous Applications 2

- Mealy machines may depend instantaneously on their inputs.
- Difficult to distribute, so require that communications between machines must be delayed.



  - ‘Synchronous Network Model’ of Distributed Systems theory
  - compute–communicate–compute–$\cdots$

- Logical delays in specification are implemented within protocol.
- Take a synchronous application: $N = m_1 \mid\mid m_2 \mid\mid \ldots \mid\mid m_p$ and place each machine m_i on a distinct network node.
- The distributed version must produce the same sequences as the synchronous version.
  - Test, simulate, reason, verify synchronous program.
  - Distributed implementation (e.g., for physical proximity to sensors and actuators).

Example

Very simple application:

$$\text{let app}() = \text{o1}, \text{o2 where}$$
$$\quad \text{rec o1} = 0 \rightarrow \text{pre}\ (\text{o2} + 2)$$
$$\quad \text{and o2} = 1 \rightarrow \text{pre}\ (\text{o1} + 2)$$

Example synchronous execution:

o1	0	3	4	7	8	11	12	···
o2	1	2	5	6	9	10	13	···

Example

Very simple application:

```
let app() = o1, o2 where
  rec o1 = 0 → pre (o2 + 2)
  and o2 = 1 → pre (o1 + 2)

 let node m1(l2) = 0 → (l2 + 2)
 let node m2(l1) = 0 → (l1 + 2)
```

Example synchronous execution:

o1	0	3	4	7	8	11	12	⋯
o2	1	2	5	6	9	10	13	⋯

Example

Very simple application:

```
let app() = o1, o2 where
  rec o1 = 0 → pre (o2 + 2)
  and o2 = 1 → pre (o1 + 2)

let node m1(l2) = 0 → (l2 + 2)
let node m2(l1) = 0 → (l1 + 2)

let node qp_app(c1, dc1, c2, dc2) = o1, o2 where
  rec present c1() → do emit o1 = m1(l2) done
  and present c2() → do emit o2 = m2(l1) done
  and l1 = link(c1, dc1, o1)
  and l2 = link(c2, dc2, o2)
```

Works correctly if $T_{\min} = T_{\max} \geq \tau_{\max}$

Example synchronous execution:

o1	0	3	4	7	8	11	12	$\cdots$
o2	1	2	5	6	9	10	13	$\cdots$

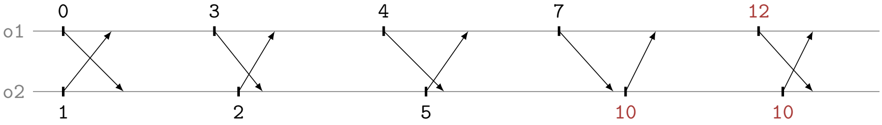
Very simple application:

```
let app() = o1, o2 where
  rec o1 = 0 → pre (o2 + 2)
  and o2 = 1 → pre (o1 + 2)

let node m1(l2) = 0 → (l2 + 2)
let node m2(l1) = 0 → (l1 + 2)

let node qp_app(c1, dc1, c2, dc2) = o1, o2 where
  rec present c1() → do emit o1 = m1(l2) done
  and present c2() → do emit o2 = m2(l1) done
  and l1 = link(c1, dc1, o1)
  and l2 = link(c2, dc2, o2)
```

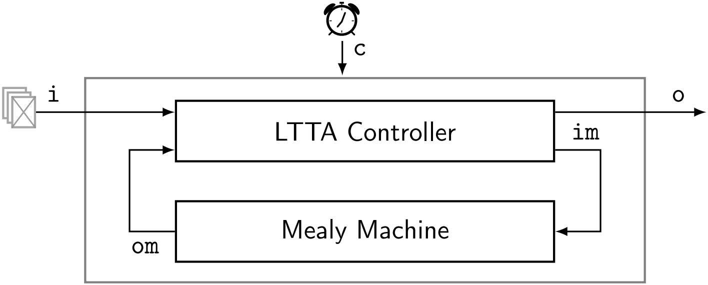
Example synchronous execution:

o1	0	3	4	7	8	11	12	⋯
o2	1	2	5	6	9	10	13	⋯

Works correctly if $T_{\min} = T_{\max} \geq \tau_{\max}$, but not otherwise



LTTA nodes



let node ltta_node(i) = o where
  rec (o, im) = ltta_controller(i, om)
  and present im(v) → do emit om = machine(v) done

val ltta_node : α list $\xrightarrow{D}$ β signal

At instants determined by the protocol, the controller samples a list of inputs to triggers the embedded machine, and controls the publication of the output.

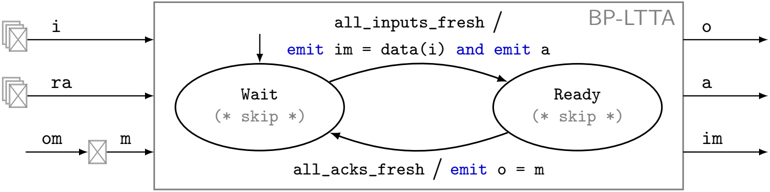
The LTTA Controller must preserve the global synchronous semantics.

- To counter oversampling: use an alternating bit.

```
type α msg = {data: α; alt: bool}

let node alternate i = o where
  rec present i(v) → local flag in
        do flag = true → not (pre flag)
        and emit o = {data = v; alt = flag} done

let node ltta_link(c, dc, i, mi) = o where
  rec s = channel(c, dc, i)
  and o = mem(alternate(s), {data = mi; alt = false})

let node fresh (input, read, initial_state) = o where
  rec init m = initial_state
  and present read(_) → do m = input.alt done
  and o = (input.alt <> last m)
```

- To counter overwriting: acknowledgement or waiting.

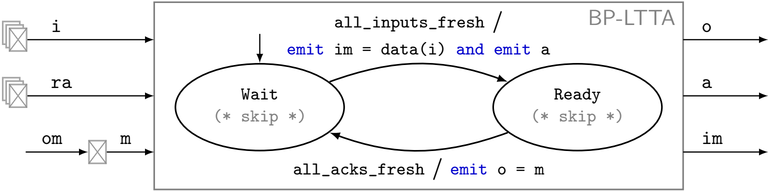
Back-Pressure LTTA



- The additional inputs ra are acknowledgements from consumers.
- The additional output a is for acknowledging producers.
- The application 'skips' until the controller is ready.

```
let node bp_controller (i, ra, om, mi) = (o, a, im) where
  rec m = mem(om, mi)
  and automaton
      | Wait →
          do (* skip *)
          unless all_inputs_fresh then
            do emit im = data(i) and emit a in Ready
      | Ready →
          do (* skip *)
          unless all_acks_fresh then
            do emit o = m in Wait

  and all_inputs_fresh = forall_fresh(i, im, true)
  and all_acks_fresh = forall_fresh(ra, o, false)
```

Back-Pressure LTTA

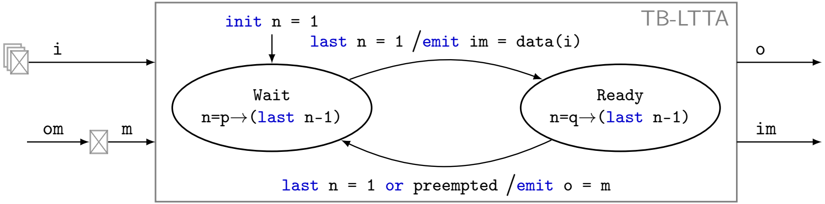


- The additional inputs ra are acknowledgements from consumers.
- The additional output a is for acknowledging producers.
- The application 'skips' until the controller is ready.

```
let node bp_controller (i, ra, om, mi) = (o, a, im) where
  rec m = mem(om, mi)
  and automaton
    | Wait →
      do (* skip *)
      unless all_inputs_fresh then
        do emit im = data(i) and emit a in Ready
    | Ready →
      do (* skip *)
      unless all_acks_fresh then
        do emit o = m in Wait

  and all_inputs_fresh = forall_fresh(i, im, true)
  and all_acks_fresh = forall_fresh(ra, o, false)
```

- Worst-case throughput: $\lambda_{\text{bp}} = 1/2(T_{\max} + \tau_{\max})$ (the maximum delay between two successive iterations is $2(T_{\max} + \tau_{\max})$)
- Nasty control dependencies.
- What about real-time behaviour?

Time-Based LTTA

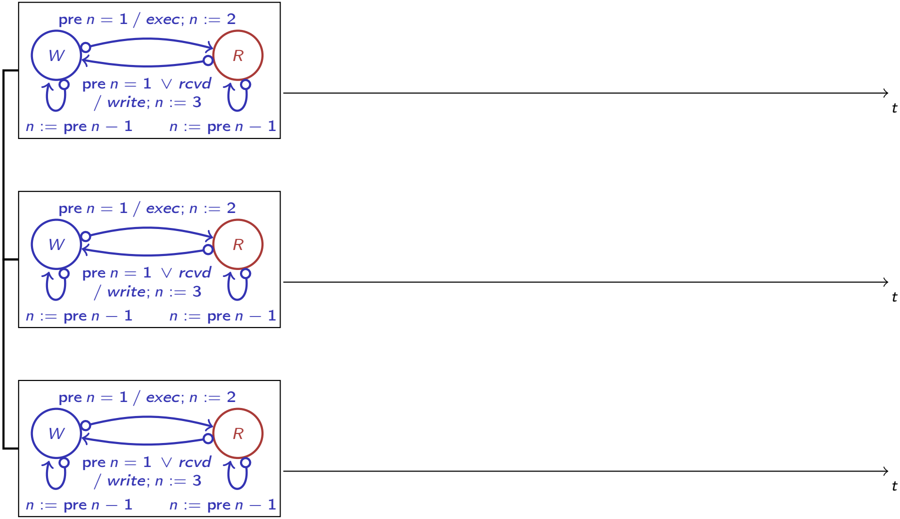


- Requires broadcast communication but not acknowledgement values.

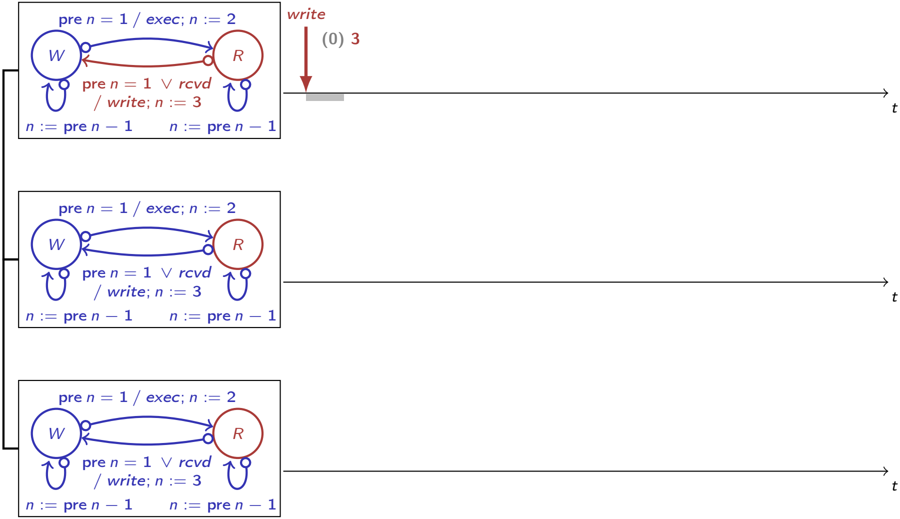
```
let node tb_controller (i, om, mi) = (o, im) where
  rec m = mem(om, mi)
  and init n = 1
  and automaton
     | Wait →
        do n = p → (last n − 1)
        unless (last n = 1) then
          do emit im = data(i) in Ready
     | Ready →
        do n = q → (last n − 1)
        unless ((last n = 1) or preempted) then
          do emit o = m in Wait

  and preempted = exists_fresh(i, im, true)
```

- Nodes alternate between write and read/execute phases.
- Correctness depends on the parameters:
  p: num. wait before sampling inputs
  q: max. wait before sending outputs
- 2-state version improves on 5-state and Petri net versions

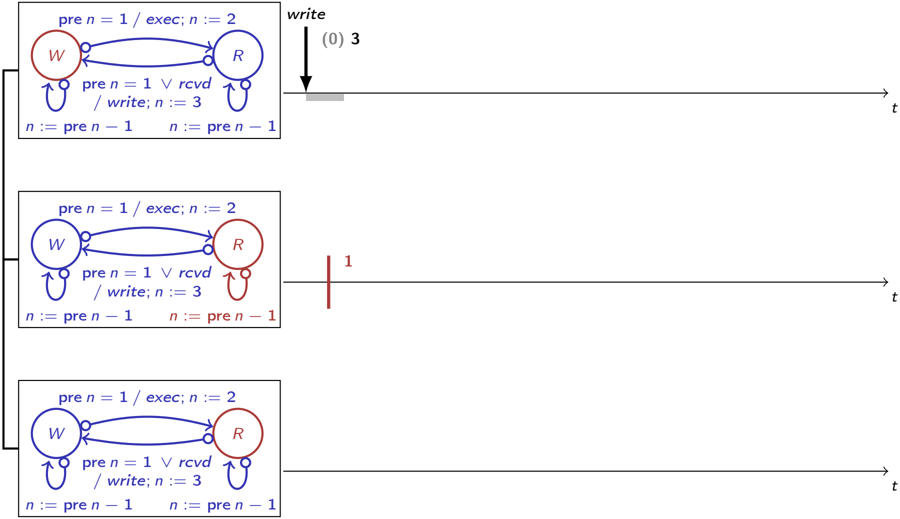
Time-Based LTTA ($p = 3, q = 2$)



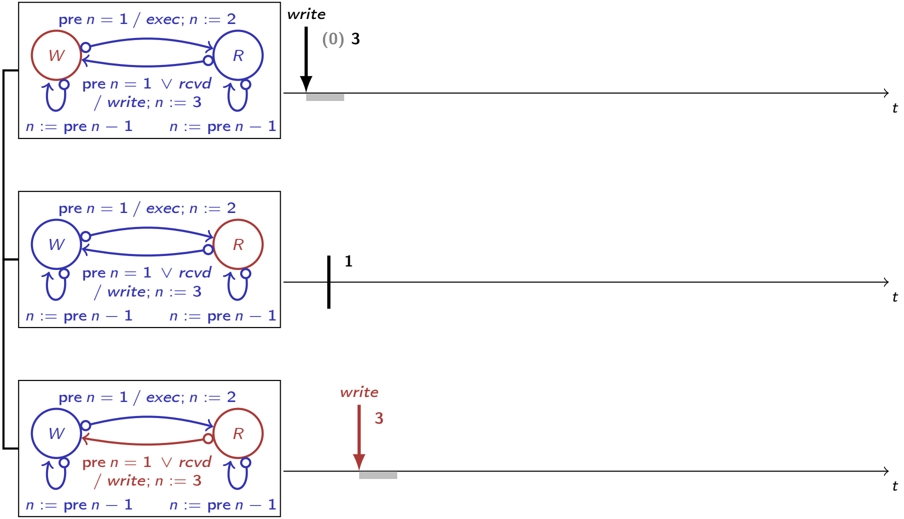
Time-Based LTTA ($p = 3, q = 2$)



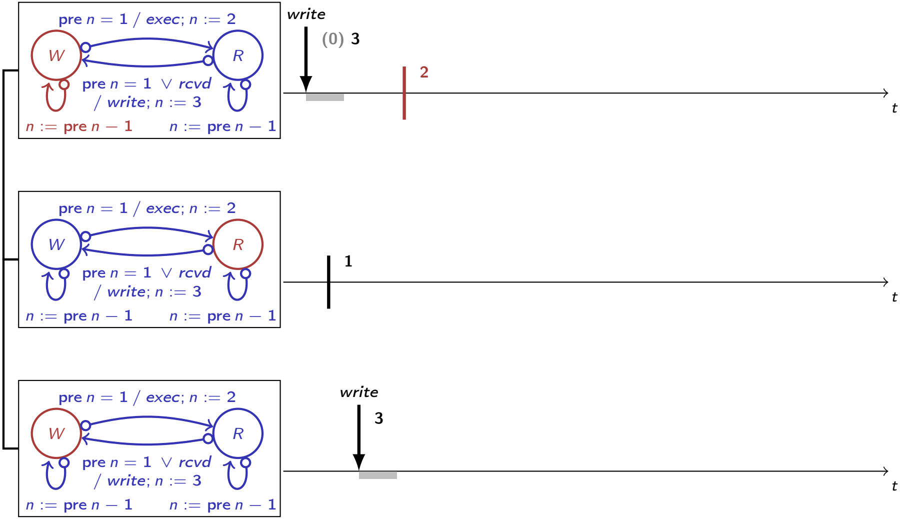
Time-Based LTTA ($p = 3, q = 2$)



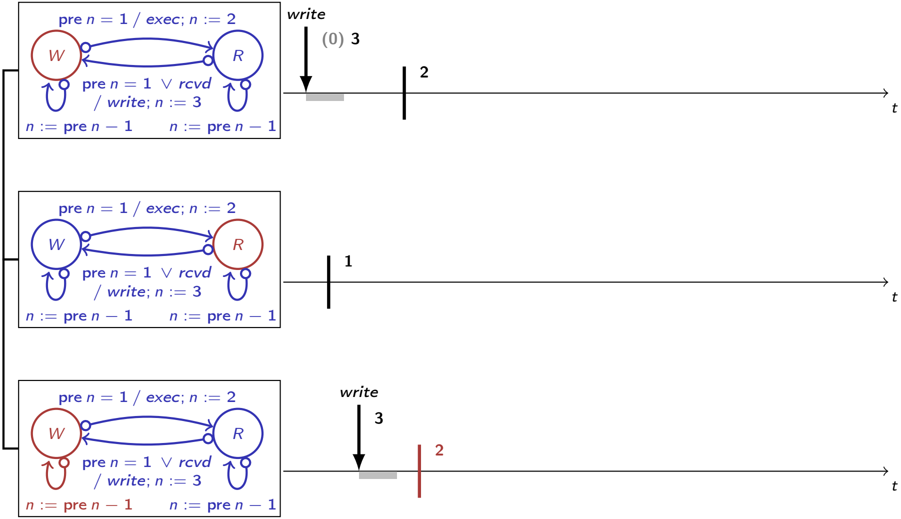
Time-Based LTTA ($p = 3, q = 2$)



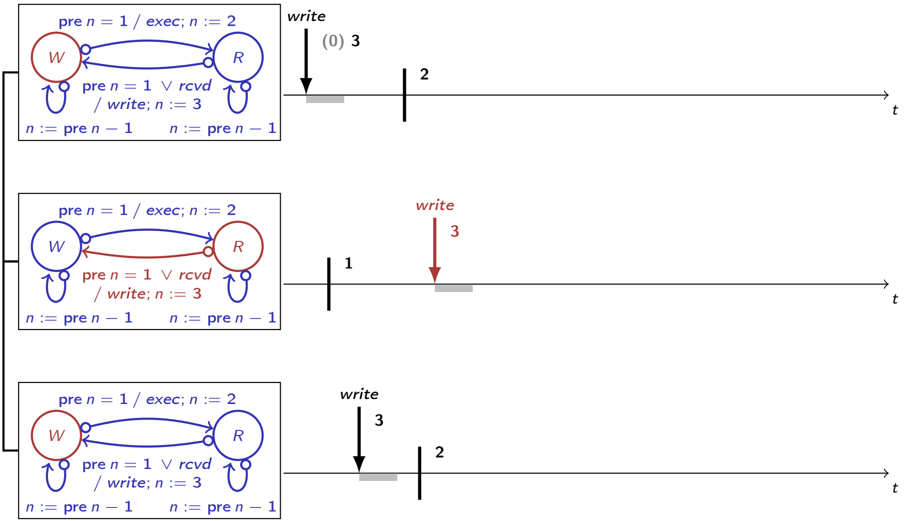
Time-Based LTTA ($p = 3, q = 2$)



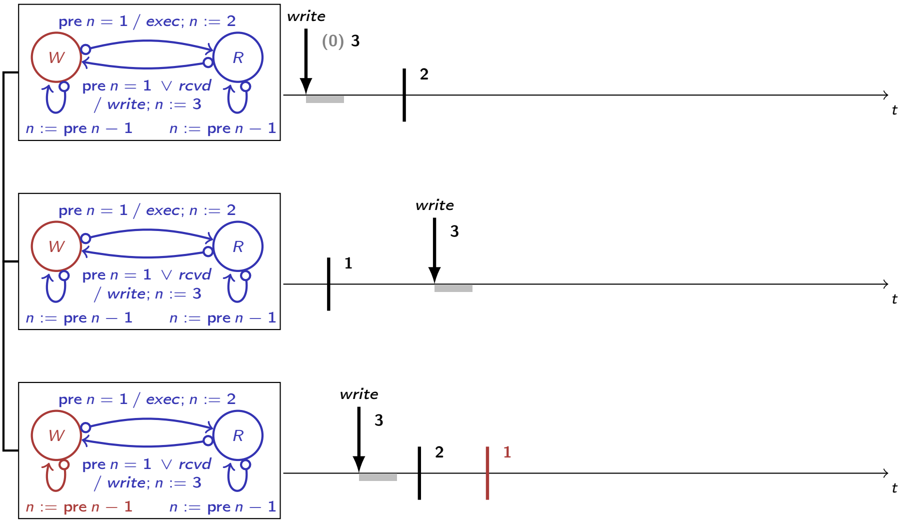
Time-Based LTTA ($p = 3, q = 2$)



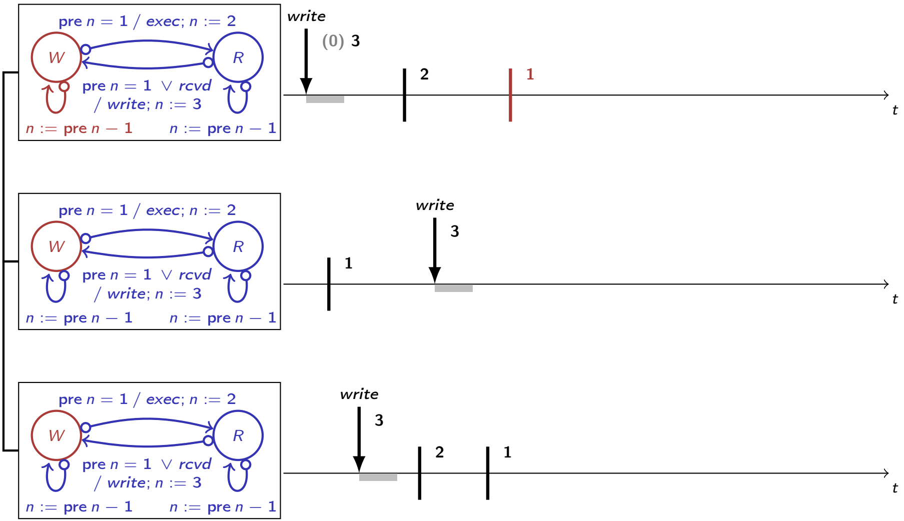
Time-Based LTTA ($p = 3, q = 2$)



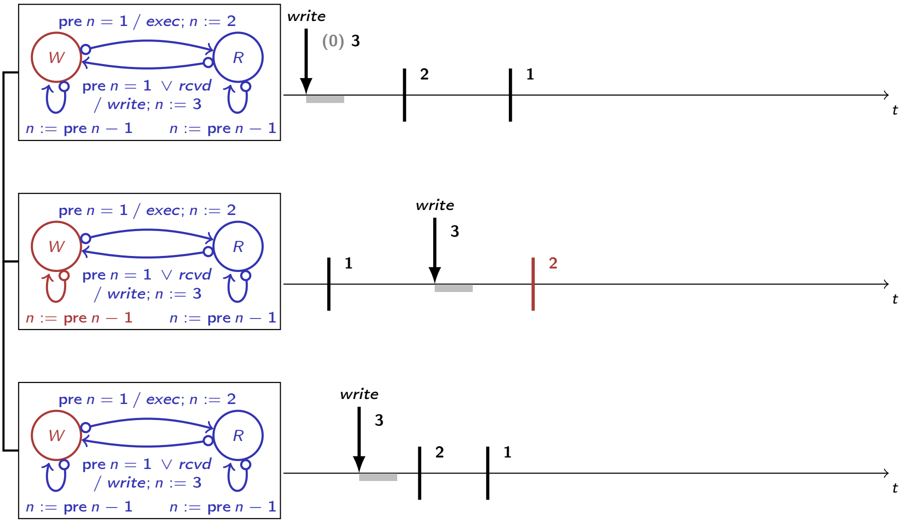
Time-Based LTTA ($p = 3, q = 2$)



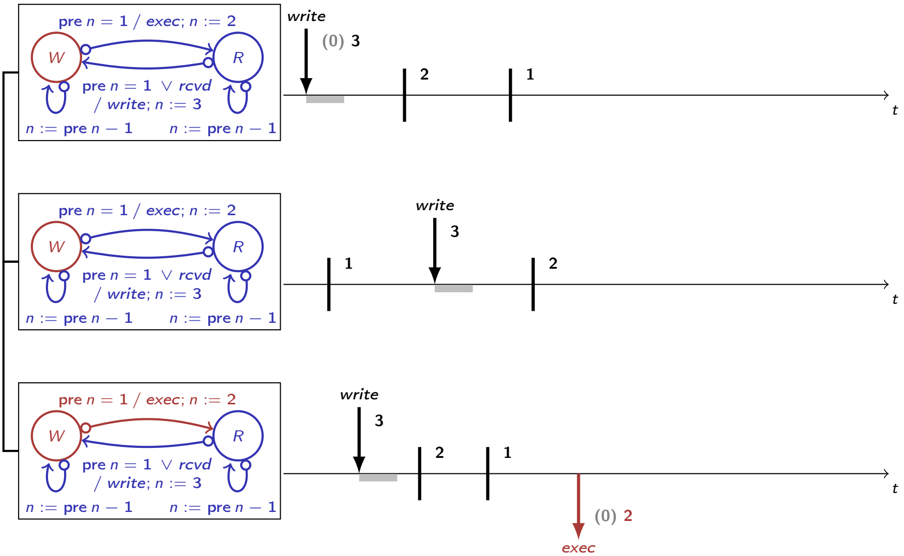
Time-Based LTTA $(p = 3, q = 2)$



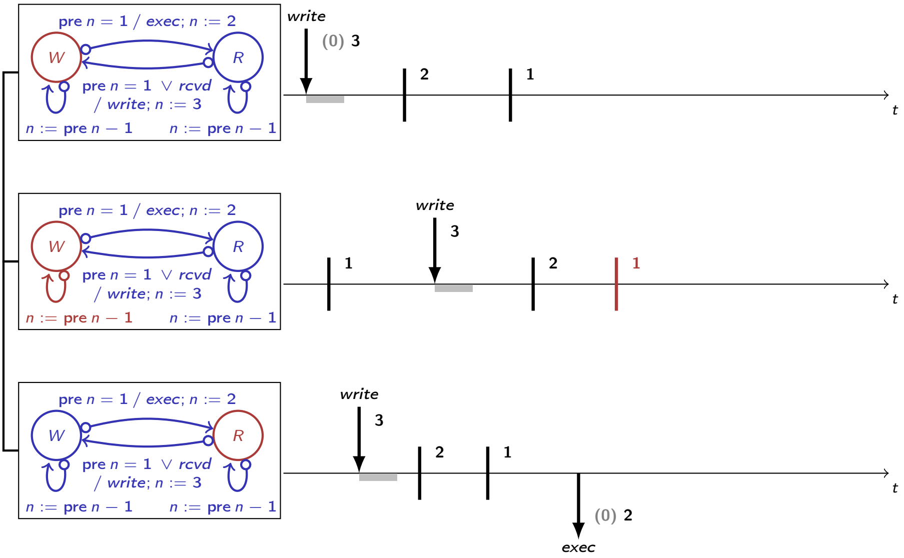
Time-Based LTTA ($p = 3, q = 2$)



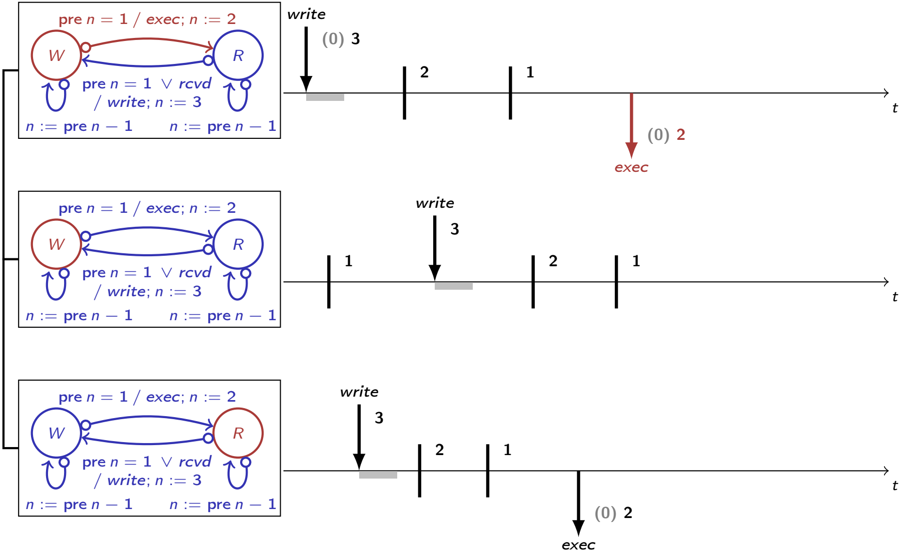
Time-Based LTTA ($p = 3, q = 2$)



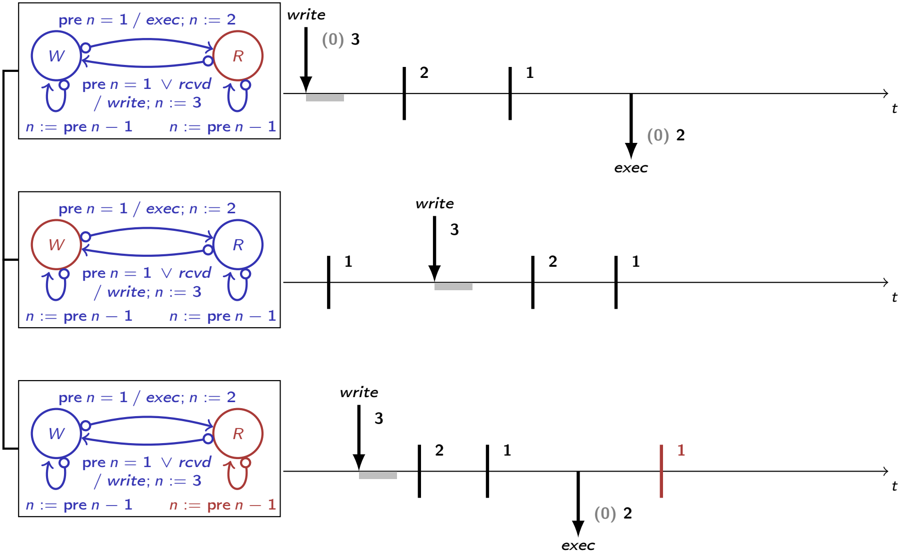
Time-Based LTTA ($p = 3, q = 2$)



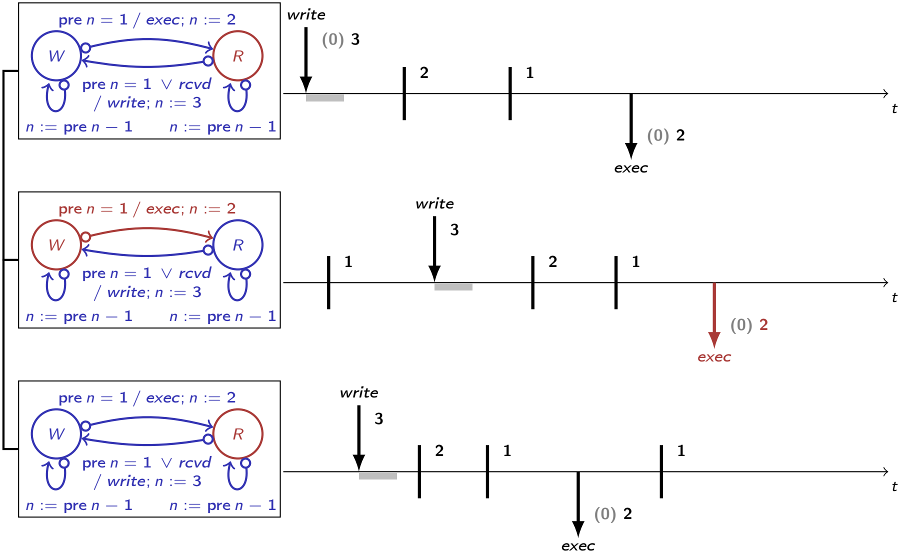
Time-Based LTTA ($p = 3, q = 2$)



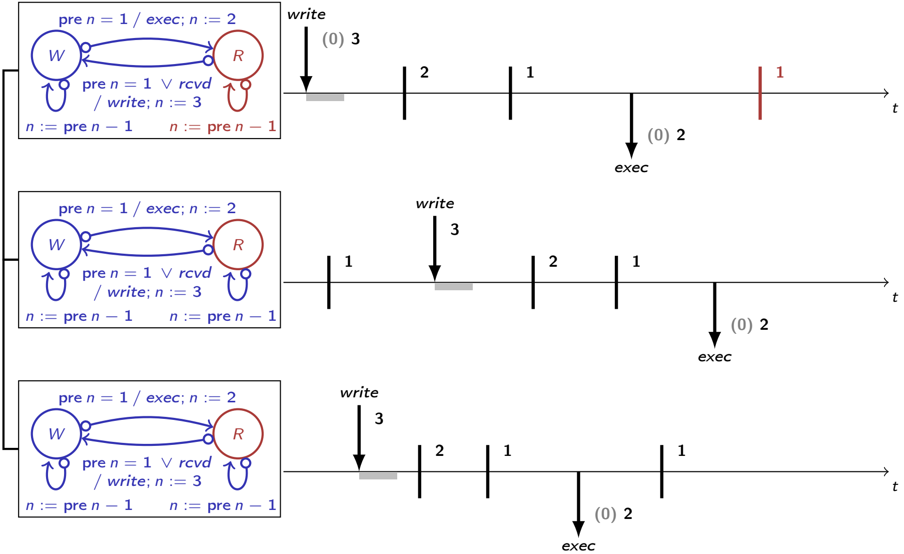
Time-Based LTTA ($p = 3, q = 2$)



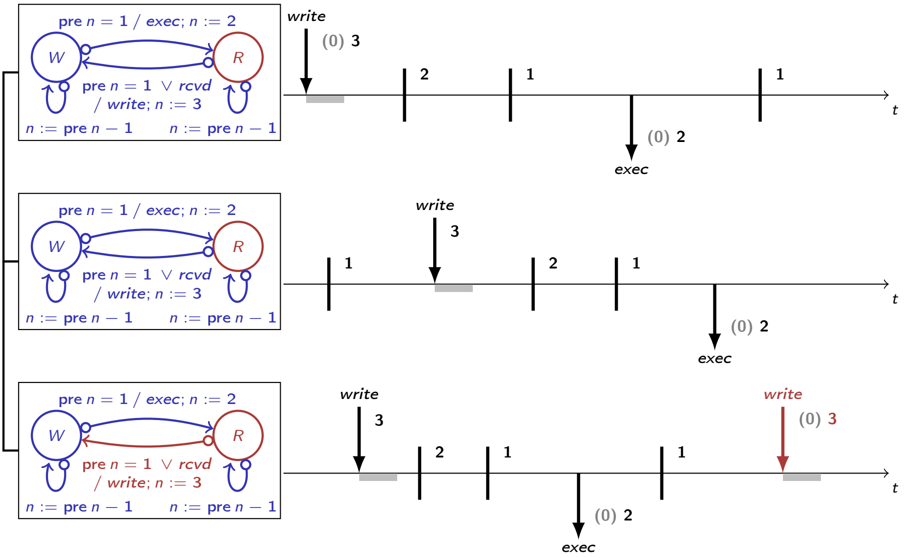
Time-Based LTTA ($p = 3, q = 2$)



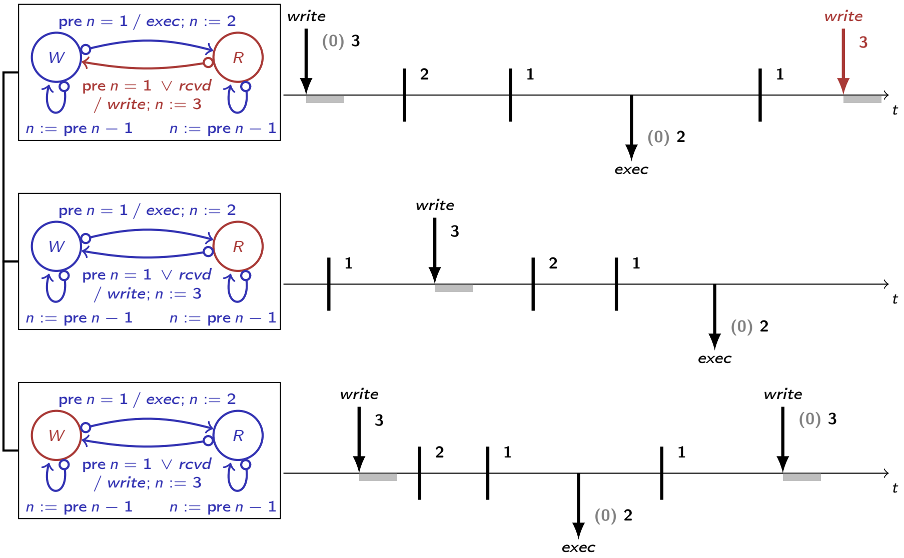
Time-Based LTTA ($p = 3, q = 2$)



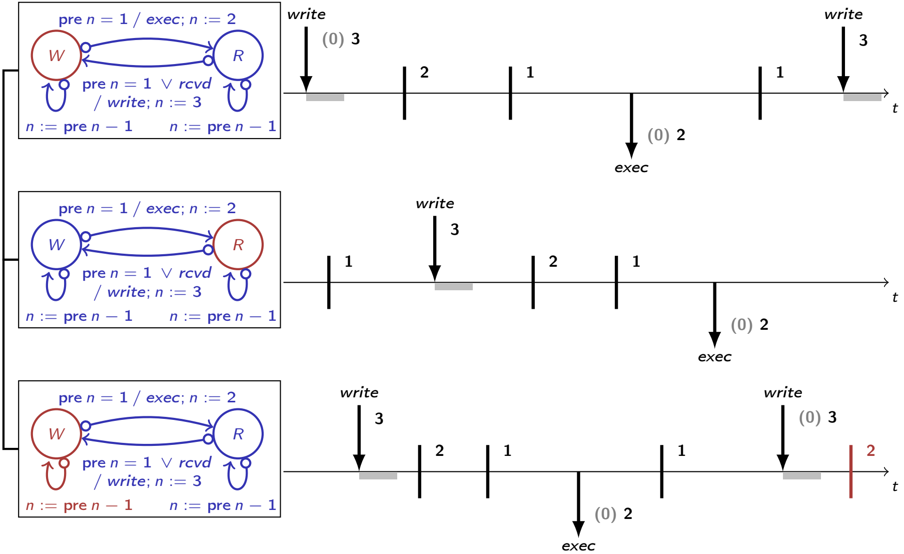
Time-Based LTTA ($p = 3, q = 2$)



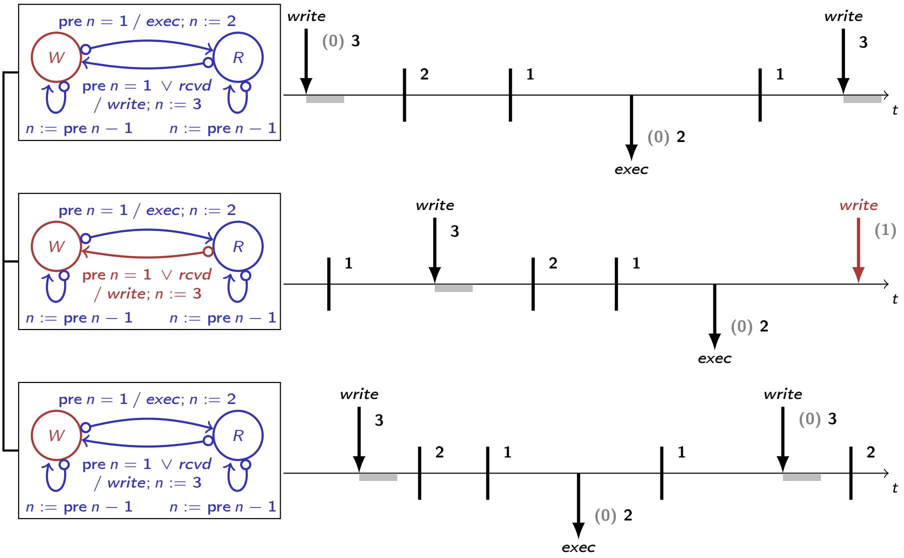
Time-Based LTTA ($p = 3, q = 2$)



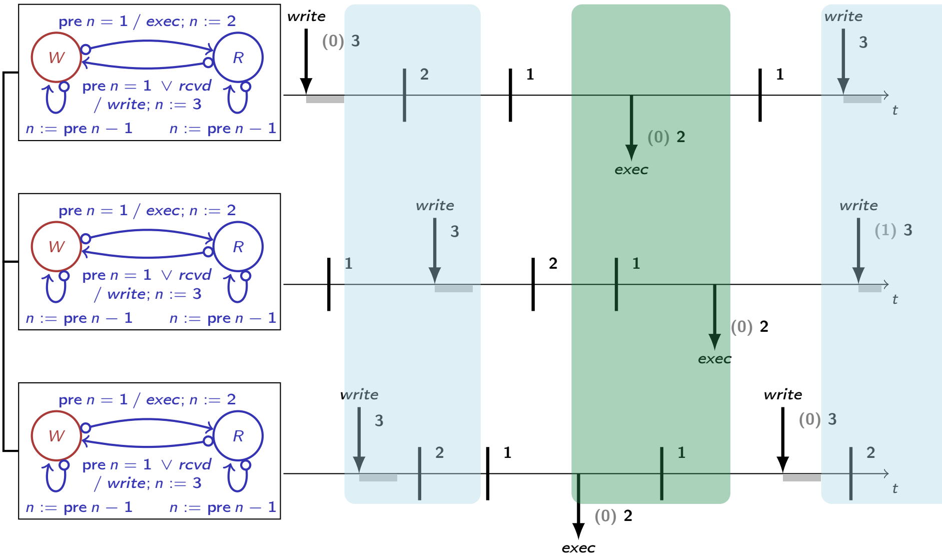
Time-Based LTTA ($p = 3, q = 2$)



Time-Based LTTA ($p = 3, q = 2$)



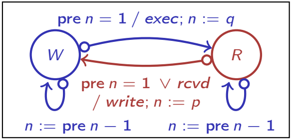
Time-Based LTTA ($p = 3, q = 2$)



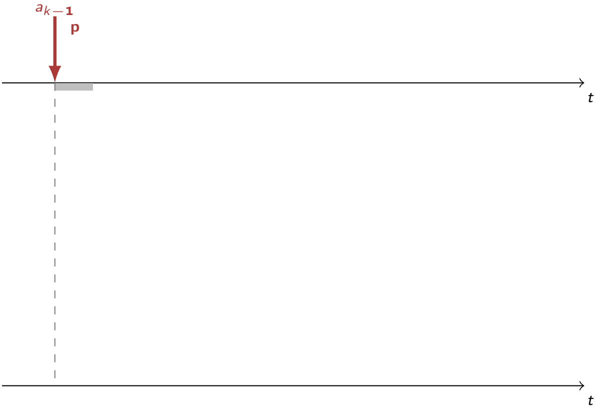

pre $n = 1$ / *exec*; $n := q$
W
R
pre $n = 1 \vee$ *rcvd* / *write*; $n := p$
$n :=$ **pre** $n - 1$
$n :=$ **pre** $n - 1$
t

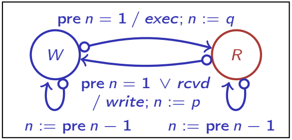

pre $n = 1$ / *exec*; $n := q$
W
R
pre $n = 1 \vee$ *rcvd* / *write*; $n := p$
$n :=$ **pre** $n - 1$
$n :=$ **pre** $n - 1$
t

Time-Based LTTA: calculation of p and q



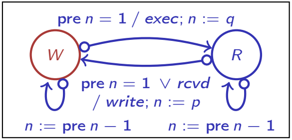
P_A ***is earliest/fastest*** $(T_A^i = T^{\min})$



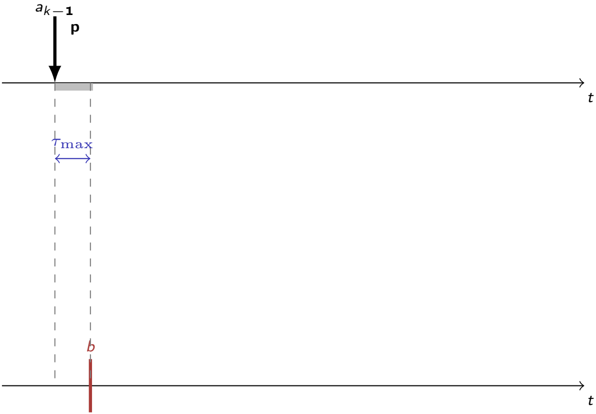


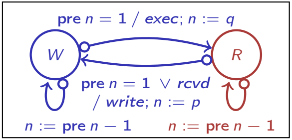
P_B ***is latest/slowest*** $(T_B^i = T^{\max})$

Time-Based LTTA: calculation of p and q



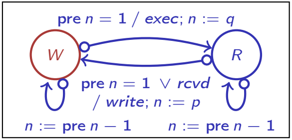
P_A ***is earliest/fastest*** $(T_A^i = T^{\min})$



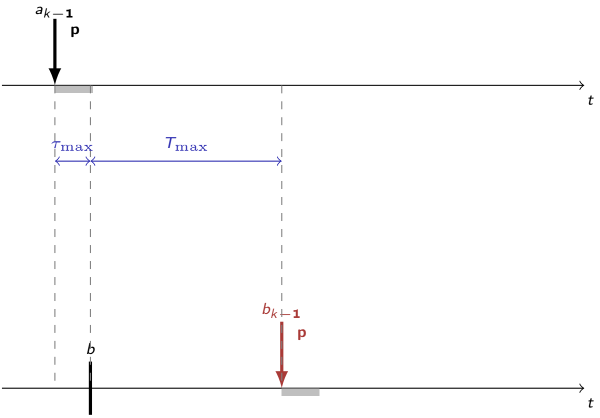


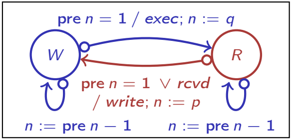
P_B ***is latest/slowest*** $(T_B^i = T^{\max})$

Time-Based LTTA: calculation of p and q



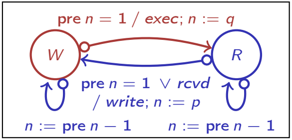
P_A is earliest/fastest ($T_A^i = T^{\min}$)



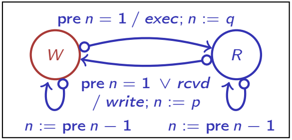


P_B is latest/slowest ($T_B^i = T^{\max}$)

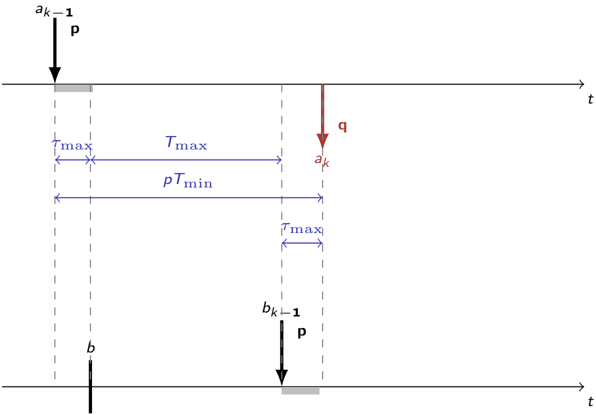
Time-Based LTTA: calculation of p and q



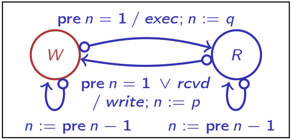
P_A ***is earliest/fastest*** $(T_A^i = T^{\min})$



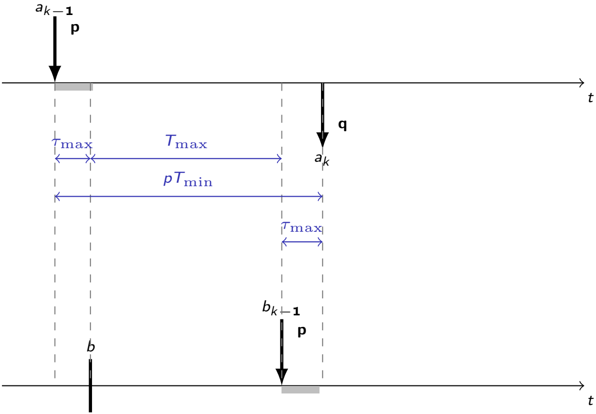
P_B ***is latest/slowest*** $(T_B^i = T^{\max})$

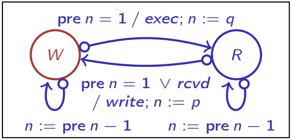


Time-Based LTTA: calculation of p and q



P_A ***is earliest/fastest*** $(T_A^i = T^{\min})$

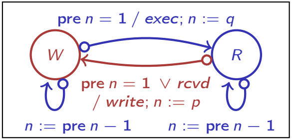




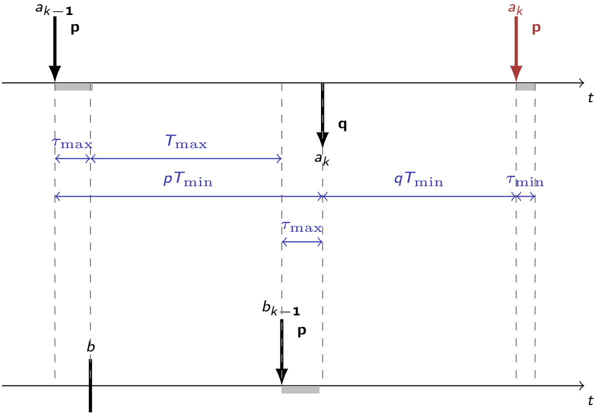
P_B ***is latest/slowest*** $(T_B^i = T^{\max})$

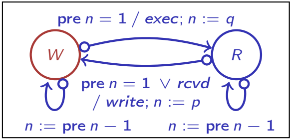
1. $pT_{\min} > \tau_{\max} + T_{\max} + \tau_{\max}$

Time-Based LTTA: calculation of p and q



P_A is earliest/fastest ($T_A^i = T^{\min}$)

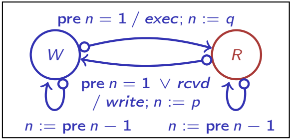




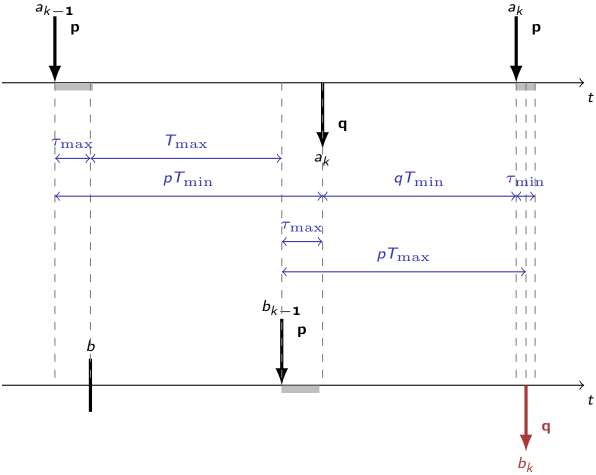
P_B is latest/slowest ($T_B^i = T^{\max}$)

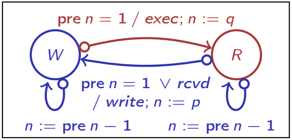
1. $pT_{\min} > \tau_{\max} + T_{\max} + \tau_{\max}$

Time-Based LTTA: calculation of p and q



P_A ***is earliest/fastest*** $(T_A^i = T^{\min})$

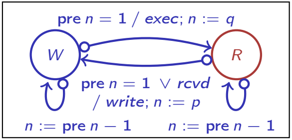




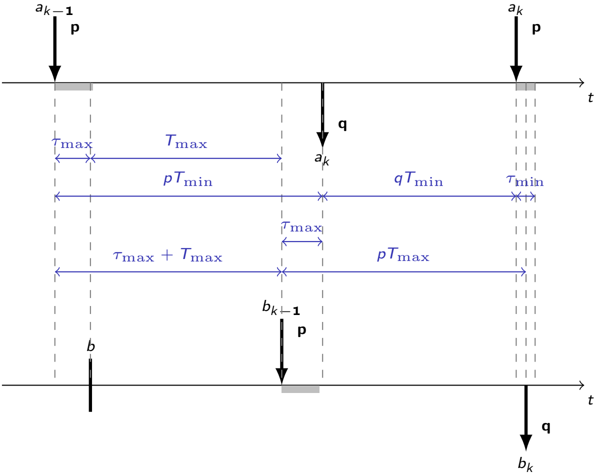
P_B ***is latest/slowest*** $(T_B^i = T^{\max})$

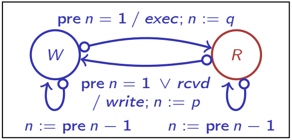
1. $pT_{\min} > \tau_{\max} + T_{\max} + \tau_{\max}$

Time-Based LTTA: calculation of p and q



P_A ***is earliest/fastest*** ($T_A^i = T^{\min}$)





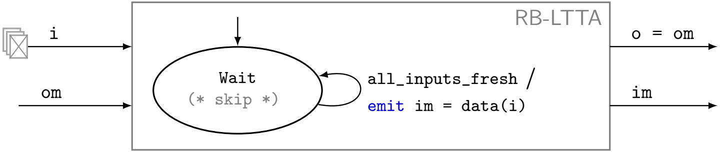
P_B ***is latest/slowest*** ($T_B^i = T^{\max}$)

1. $pT_{\min} > \tau_{\max} + T_{\max} + \tau_{\max}$
2. $pT_{\min} + qT_{\min} + \tau_{\min} > \tau_{\max} + T_{\max} + pT_{\max}$

Time-Based LTTA: evaluation

- The worst-case throughput is $\lambda_{\mathsf{tb}} = 1/(p^* + q^*)\,T_{\max}$
  (where p^* and q^* are optimal values).
  The slowest node spends $p^*\,T_{\max}$ in Wait and $q^*\,T_{\max}$ in Ready.
- Less efficient than Back-Pressure LTTA, but no control dependencies.
- Less efficient than clock synchronization, but simpler implementation.

Round-Based LTTA



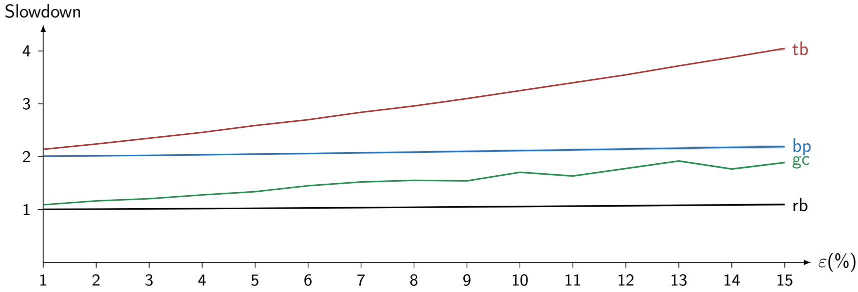
- Requires broadcast communication but not acknowledgement values.

```
let node rb_controller(i, om) = (o, im) where
  rec automaton
      | Wait →
        do (* skip *)
        unless all_inputs_fresh then
          do emit im = data(i) in Wait
  and all_inputs_fresh = forall_fresh(i, im, true)
  and o = om
```

- Simplification of back-pressure idea under broadcast assumption.
- No alternation between write and read/execute phases.
- Communicate via buffers of size 2.
- Worst-case throughput: $\lambda_{\text{bp}} = 1/(T_{\max} + \tau_{\max})$
- What if a node or communication link fails?

Slowdown factor for 2-node application (smaller = better)

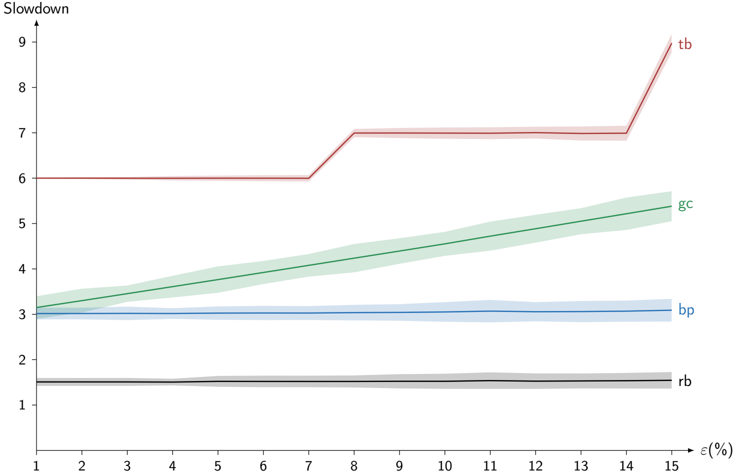
$T \ll \tau$, $T_{\mathrm{nom}}/\tau_{\mathrm{nom}} = 0.01$



- Averaged simulation results on simple application.
- Relative worst-case slowdowns versus jitter
- Round-based protocol close to optimum.
- Back-pressure is twice as slow due to dual mode operation.

Slowdown factor for 2-node application (smaller = better)

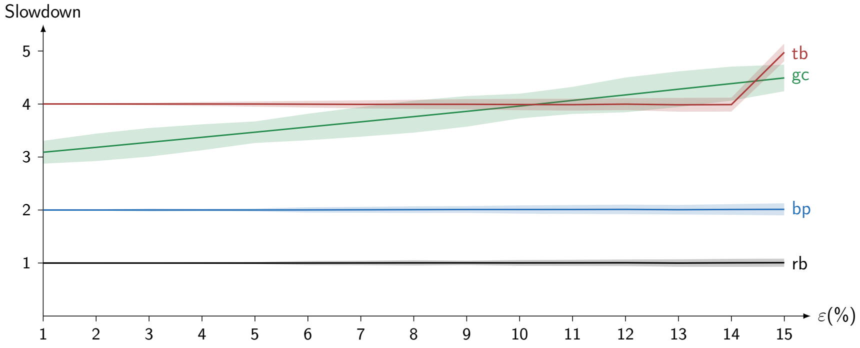
$T \approx \tau,\ T_{\text{nom}}/\tau_{\text{nom}} = 1$



- Time-based: steep changes due to integer components.
- Time-based and global clock: both sensitive to jitter.

Slowdown factor for 2-node application (smaller = better)

$T \gg \tau,\ T_{\mathrm{nom}}/\tau_{\mathrm{nom}} = 100$



- Clock synchronization overhead is more significant at larger activation periods.
- Time-based protocol is nearly always slower (no pipelining, pessimistic operation).

LTTA: Open questions

- Machine-assisted proof of parameterized LTTA systems from Zélus models?
- Reasoning about the effect of protocol on real-time applications?

Outline

The synchronous language Lustre

[Caspi, Pilaud, Halbwachs, and Plaice (1987): LUSTRE: A declarative language for programming synchronous systems]

- Ideal for programming an important class of embedded controllers.
  - Academic foundation of Scade Suite tool for critical industrial systems.

- Based on a discrete-time abstraction.

```
every trigger:
  read inputs;
  compute;
  write outputs
```

model: R_1, R_2, R_3, R_4, R_5, ...

The synchronous language Lustre

[Caspi, Pilaud, Halbwachs, and Plaice (1987): LUSTRE: A declarative language for programming synchronous systems]

- Ideal for programming an important class of embedded controllers.
  - Academic foundation of Scade Suite tool for critical industrial systems.

- Based on a discrete-time abstraction.

```
every trigger:
  read inputs;
  compute;
  write outputs
```

model: R_1, R_2, R_3, R_4, R_5, ...

But, 'physical' timing constraints are often required.

The synchronous language Lustre [Caspi, Pilaud, Halbwachs, and Plaice (1987): LUSTRE: A declarative language for programming synchronous systems]

- Ideal for programming an important class of embedded controllers.
  - » Academic foundation of Scade Suite tool for critical industrial systems.

- Based on a discrete-time abstraction.

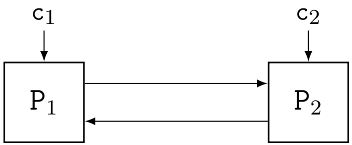
```
every trigger:
  read inputs;
  compute;
  write outputs
```

model: R_1, R_2, R_3, R_4, R_5, ...

But, 'physical' timing constraints are often required.

Timed (Safety) Automata [Alur and Dill (1994): A Theory of Timed Automata] [Henzinger, Nicollin, Sifakis, and Yovine (1994): Symbolic Model Checking for Real-Time Systems]

- Model the passage of time and timing non-determinism
  - » (tolerances in requirements / uncertainties in implementations).
- Verification and Symbolic Simulation in Uppaal [Behrmann, David, Larsen, Håkansson, Pettersson, Yi, and Hendriks (2006): Uppaal 4.0]

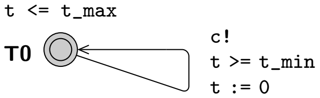


Two network nodes activated on clock inputs c_1 and c_2

- Each node is periodically triggered by a local clock.
- The difference between ticks i and $i+1$ is bounded:

$$T_{\min} \leq t_{i+1} - t_i \leq T_{\max}$$

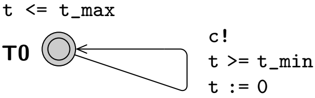
- Easy to model a clock as a Timed Automaton: [Vaandrager and Groot (2006): Analysis of a Biphase Mark Protocol with Uppaal and PVS]



- What about combining with discrete controller code?

Clock in Zélus?

```
let hybrid clock(t_min, t_max) = c where
  rec der t = 1.0 init 0.0 reset c() → 0.0
  and present up(t −. t_min) → do emit c done
```

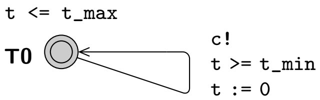


Programming Timed Automaton in Zélus

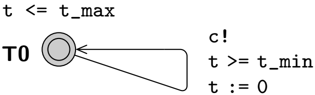
- Very restricted ODEs ($\dot{x} = 1$): no need for a numeric solver.
- Cannot express 'timing non-determinism'.

- Very appealing to 'embed' discrete programs in continuous time.
- The discrete/continuous type system rejects meaningless compositions.

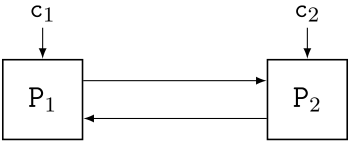
```
let hybrid clock(t_min, t_max) = c where
  rec timer t init 0 reset c() → 0
  and emit c when {t >= t_min}
  and always {t <= t_max}
```



```
let hybrid clock(t_min, t_max) = c where
  rec timer t init 0 reset c() → 0
  and emit c when {t >= t_min}
  and always {t <= t_max}
```




```
let hybrid scheduler(t_min, t_max) = c1, c2 where
  rec c1 = clock(t_min, t_max)
  and c2 = clock(t_min, t_max)
```




```
let hybrid quasinodes(t_min, t_max) = o1, o2 where
  rec c1, c2 = scheduler(t_min, t_max)
  and o1 = present c1 → node1(channel(o2)) init oi
  and o2 = present c2 → node2(channel(o1)) init oi
```

Syntax

$$
\begin{array}{rcl}
d & ::= & \texttt{let hybrid } f\texttt{(}p\texttt{) = } e \\
& | & \texttt{let node } f\texttt{(}p\texttt{) = } e \\
& | & \texttt{let } f\texttt{(}p\texttt{) = } e \\
& | & d\ d \\
e & ::= & x \mid v \mid op(e) \\
& | & \texttt{(}e\texttt{, }e\texttt{)} \\
& | & f\texttt{(}e\texttt{)} \\
& | & e\ \texttt{fby}\ e \\
& | & e\ \texttt{where rec}\ E \\
E & ::= & x\ \texttt{=}\ e \\
& | & E\ \texttt{and}\ E \\
& | & x\ \texttt{= present}\ h\ \texttt{init}\ e \\
& | & x\ \texttt{= present}\ h\ \texttt{else}\ e \\
& | & \texttt{timer}\ x\ \texttt{init}\ e\ \texttt{reset}\ h \\
& | & \texttt{always \{ } c \texttt{ \}} \\
& | & \texttt{present \{ } c \texttt{ \}} \to \texttt{do emit } x \texttt{ done}
\end{array}
$$

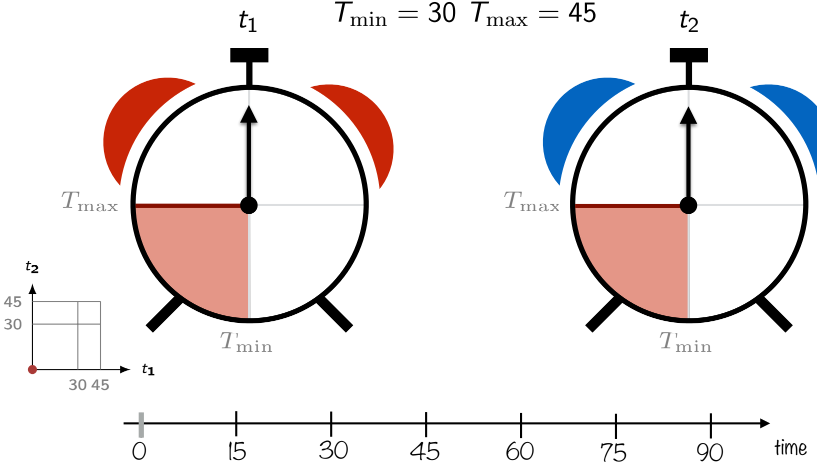
- A program is a list of declarations.
- A node is defined by an expression.
- Expressions refer to sets of equations.

New features

- Timers (time elapsing)
- Invariants (must)
- Guards (may)

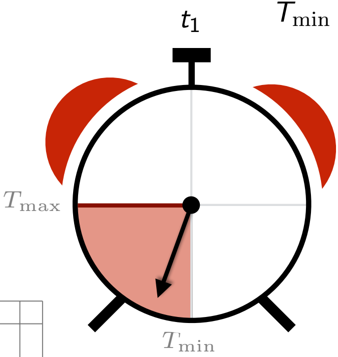
$$
\begin{array}{rcl}
p & ::= & x \mid \texttt{(}p\texttt{, }p\texttt{)} \\
h & ::= & e \to e \mid \ \ldots\ \mid e \to e \\
c & ::= & \Delta \sim e \mid c\ \texttt{\&\&}\ c \\
\Delta & ::= & x \mid x - x \\
\sim & ::= & < \mid \leq \mid \geq \mid >
\end{array}
$$

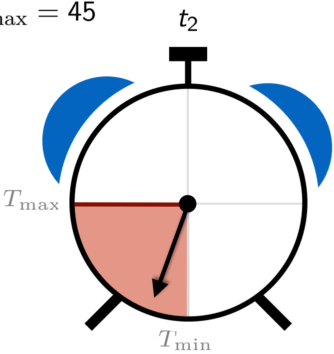
$T_{\mathrm{min}} = 30 \;\; T_{\mathrm{max}} = 45$

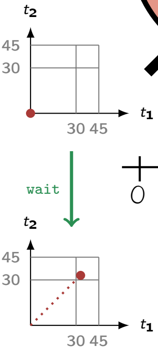


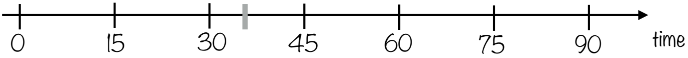
Concrete Simulation Trace

$T_{\min} = 30 \;\; T_{\max} = 45$



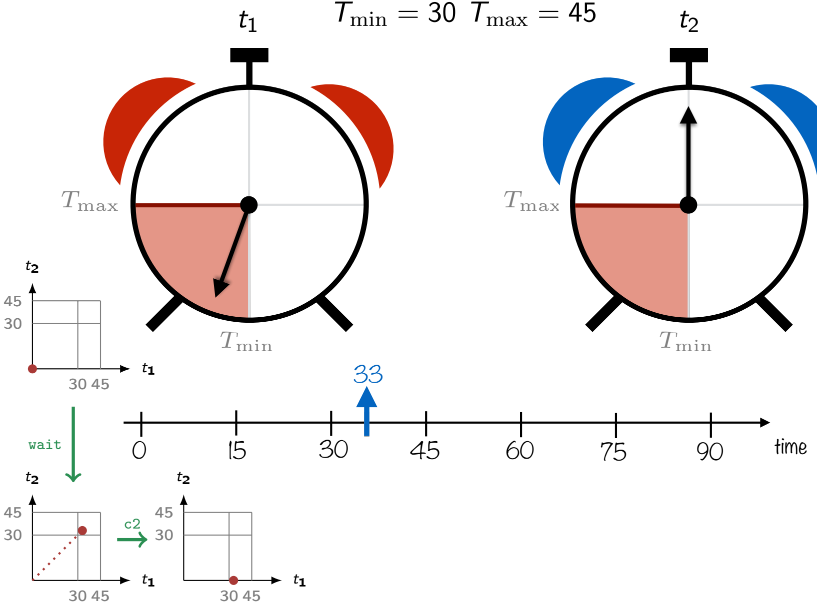






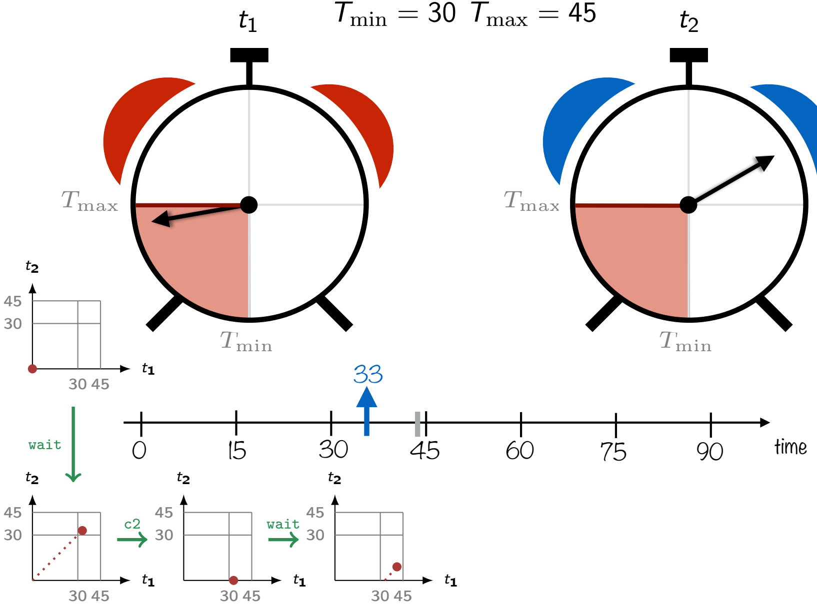
Concrete Simulation Trace

$T_{\min} = 30 \;\; T_{\max} = 45$

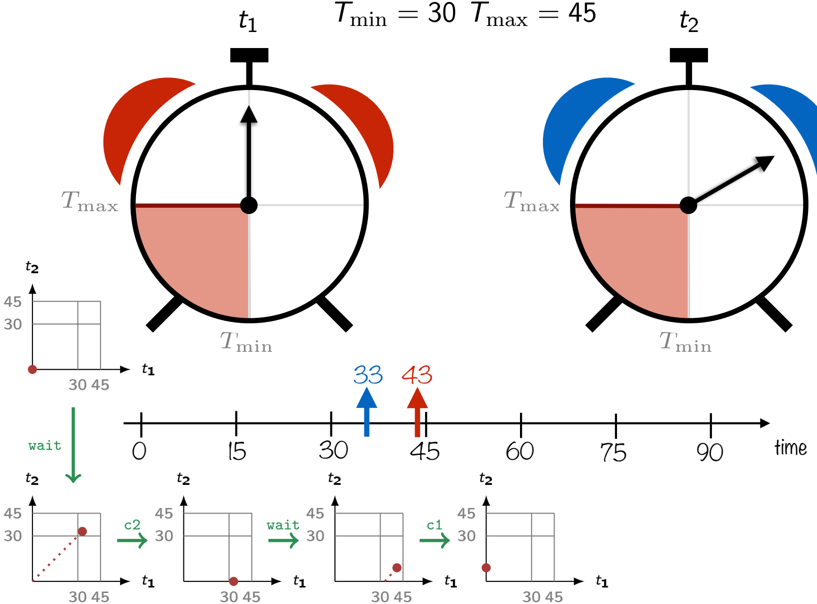


Concrete Simulation Trace

$T_{\min} = 30 \;\; T_{\max} = 45$

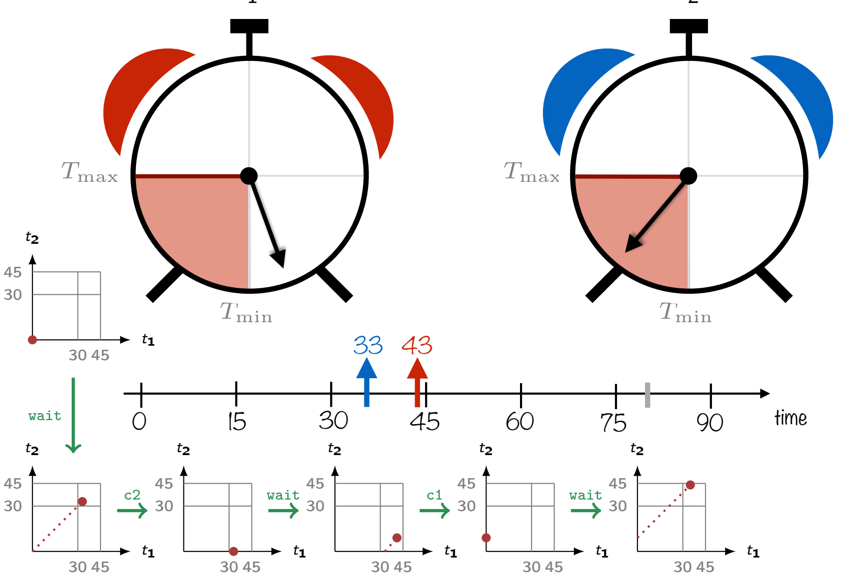


$T_{\min} = 30 \quad T_{\max} = 45$



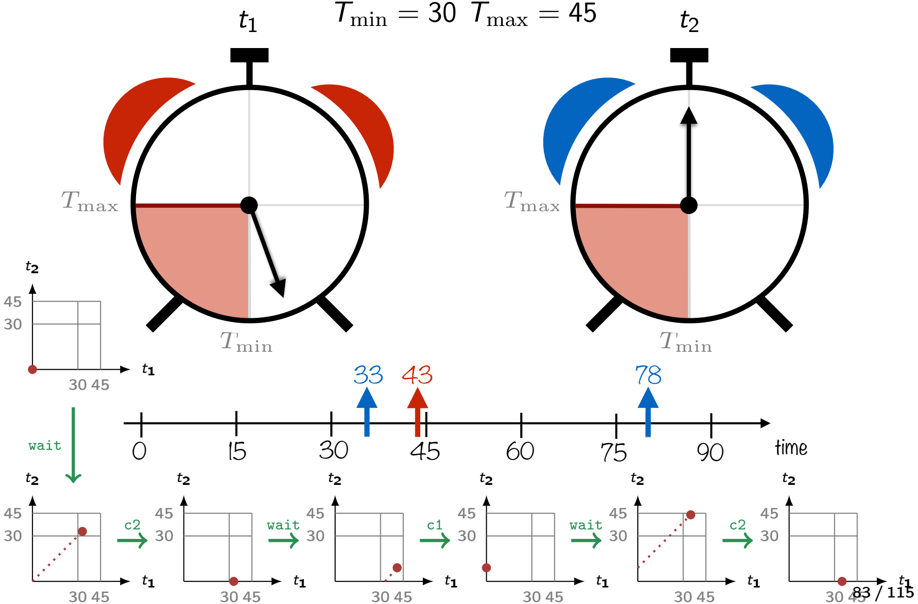
Concrete Simulation Trace

$T_{\min} = 30 \;\; T_{\max} = 45$



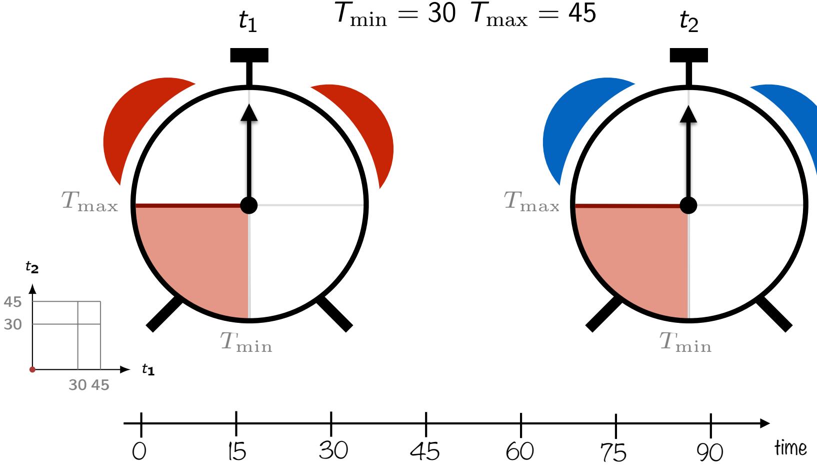
Concrete Simulation Trace

$T_{\min} = 30 \quad T_{\max} = 45$



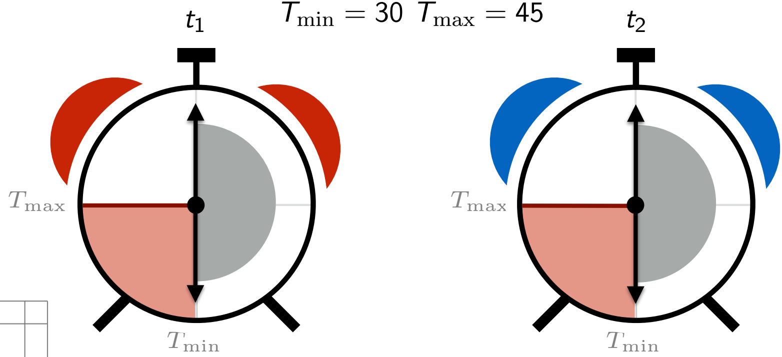
Symbolic Simulation Trace

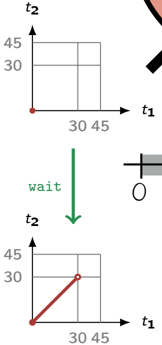
$T_{\min} = 30 \;\; T_{\max} = 45$

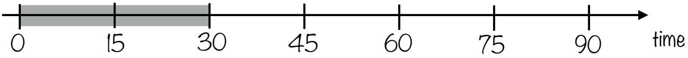


Symbolic Simulation Trace

$T_{\min} = 30 \;\; T_{\max} = 45$

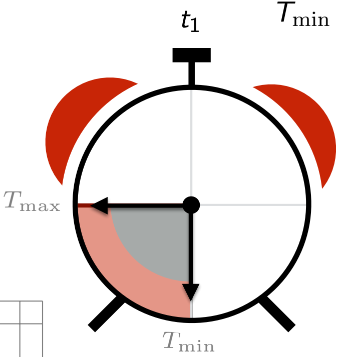


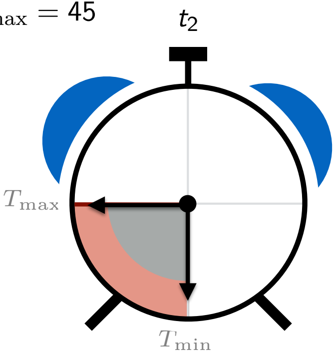


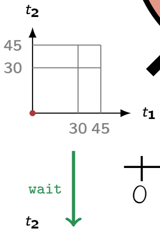


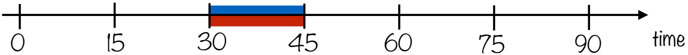
Symbolic Simulation Trace

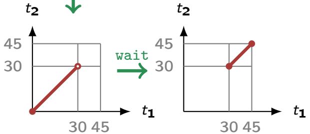
$T_{\min} = 30 \quad T_{\max} = 45$



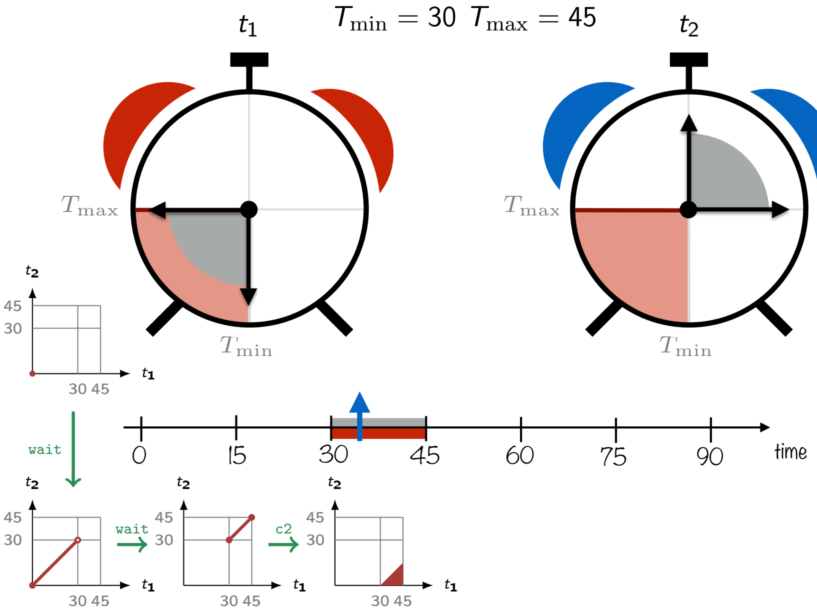






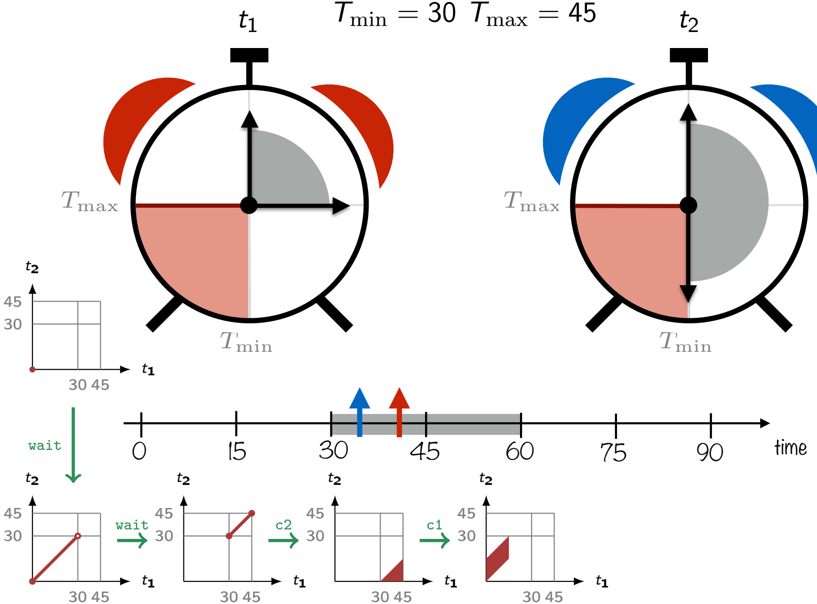


Symbolic Simulation Trace

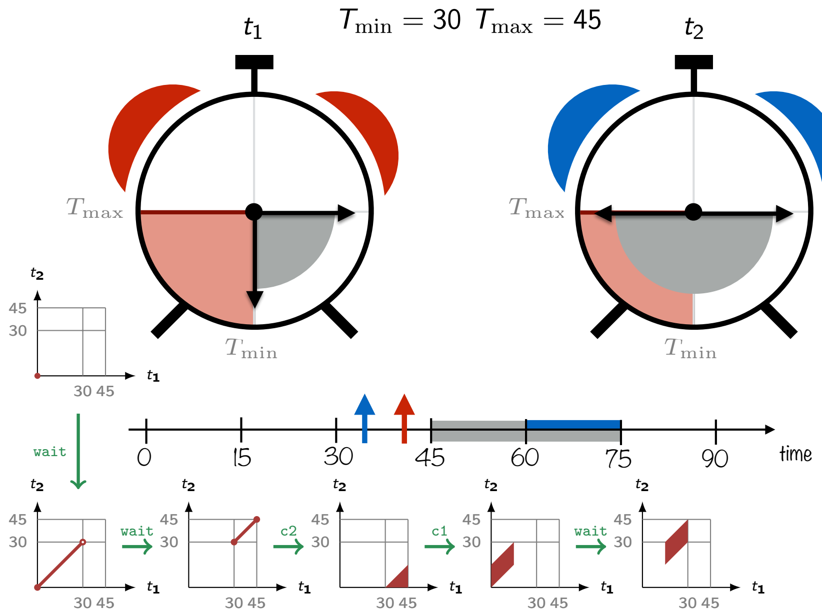


Symbolic Simulation Trace

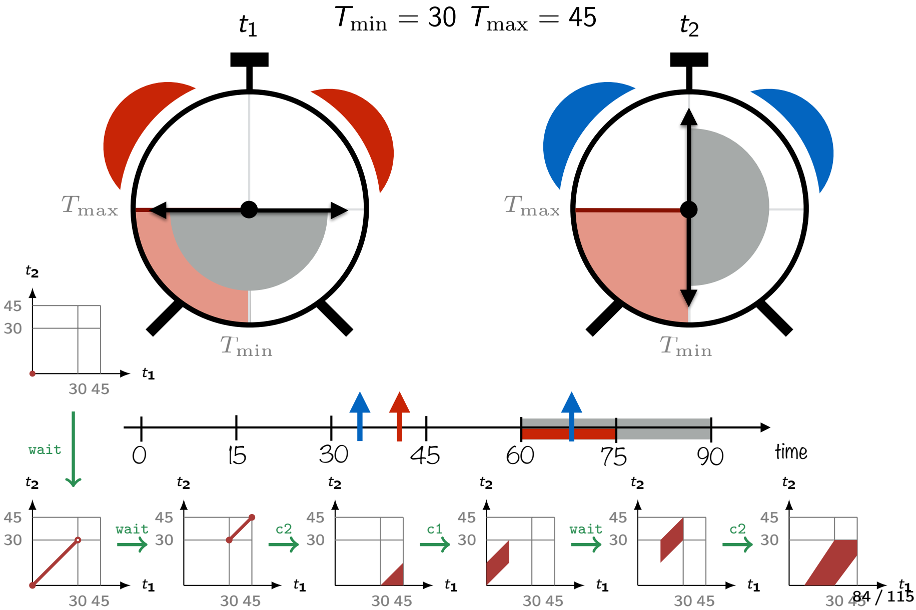
$T_{\min} = 30 \quad T_{\max} = 45$



Symbolic Simulation Trace



Symbolic Simulation Trace



Difference Bound Matrices [Dill (1989): Timing assumptions and verification of finite-state concurrent systems]

Set of constraints

$$\left\{ \begin{array}{c} t_1 < 20 \\ 6 \le t_2 \\ 5 < t_3 \le 12 \\ 4 \le t_1 - t_2 \le 8 \end{array} \right\}$$

The corresponding DBM

$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} (0,\le) & (0,\le) & (-6,\le) & (-5,<) \\ (20,<) & (0,\le) & (8,\le) & (\infty,<) \\ (\infty,<) & (-4,\le) & (0,\le) & (\infty,<) \\ (12,\le) & (\infty,<) & (\infty,<) & (0,\le) \end{bmatrix} \end{array}$$

- Represents a set of possible clock values.
- Two-dimensional array of *difference constraints*: $t_i - t_j \preceq n$ where $\preceq \in \{<, \le\}$ and $n \in \mathbb{Z} \cup \{\infty\}$.
- One dimension for each clock in the system.
  - » row = upper bounds on differences with other clocks.
  - » column = lower bounds on differences with other clocks.
- The t_0 clock is always equal to zero (for lower and upper bounds).

Difference Bound Matrices

Set of constraints

$$\left\{\begin{array}{c} t_1 < 20 \\ 6 \leq t_2 \\ 5 < t_3 \leq 12 \\ 4 \leq t_1 - t_2 \leq 8 \end{array}\right\}$$

The corresponding DBM

$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} (0,\leq) & (0,\leq) & (-6,\leq) & (-5,<) \\ (20,<) & (0,\leq) & (8,\leq) & (\infty,<) \\ (\infty,<) & (-4,\leq) & (0,\leq) & (\infty,<) \\ (12,\leq) & (\infty,<) & (\infty,<) & (0,\leq) \end{bmatrix} \end{array}$$

- Represents a set of possible clock values.
- Two-dimensional array of *difference constraints*: $t_i - t_j \preceq n$ where $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z} \cup \{\infty\}$.
- One dimension for each clock in the system.
  - row = upper bounds on differences with other clocks.
  - column = lower bounds on differences with other clocks.
- The t_0 clock is always equal to zero (for lower and upper bounds).

Set of constraints

$$\left\{ \begin{array}{c} t_1 < 20 \\ 6 \leq t_2 \\ 5 < t_3 \leq 12 \\ 4 \leq t_1 - t_2 \leq 8 \end{array} \right\}$$

The corresponding DBM

$$\begin{array}{cc} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} (0, \leq) & (0, \leq) & (-6, \leq) & (-5, <) \\ (20, <) & (0, \leq) & (8, \leq) & (\infty, <) \\ (\infty, <) & (-4, \leq) & (0, \leq) & (\infty, <) \\ (12, \leq) & (\infty, <) & (\infty, <) & (0, \leq) \end{bmatrix} \end{array}$$

- Represents a set of possible clock values.
- Two-dimensional array of *difference constraints*: $t_i - t_j \preceq n$ where $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z} \cup \{\infty\}$.
- One dimension for each clock in the system.
  - row = upper bounds on differences with other clocks.
  - column = lower bounds on differences with other clocks.
- The t_0 clock is always equal to zero (for lower and upper bounds).

Set of constraints

$$\begin{Bmatrix} t_1 < 20 \\ 6 \leq t_2 \\ 5 < t_3 \leq 12 \\ 4 \leq t_1 - t_2 \leq 8 \end{Bmatrix}$$

The corresponding DBM

$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} (0,\leq) & (0,\leq) & (-6,\leq) & (-5,<) \\ (20,<) & (0,\leq) & (8,\leq) & (\infty,<) \\ (\infty,<) & (-4,\leq) & (0,\leq) & (\infty,<) \\ (12,\leq) & (\infty,<) & (\infty,<) & (0,\leq) \end{bmatrix} \end{array}$$

- Represents a set of possible clock values.
- Two-dimensional array of *difference constraints*: $t_i - t_j \preceq n$ where $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z} \cup \{\infty\}$.
- One dimension for each clock in the system.
  - » row = upper bounds on differences with other clocks.
  - » column = lower bounds on differences with other clocks.
- The t_0 clock is always equal to zero (for lower and upper bounds).

Difference Bound Matrices [Dill (1989): Timing assumptions and verification of finite-state concurrent systems]

Set of constraints

$$\left\{\begin{array}{c} t_1 < 20 \\ 6 \leq t_2 \\ 5 < t_3 \leq 12 \\ 4 \leq t_1 - t_2 \leq 8 \end{array}\right\}$$

The corresponding DBM

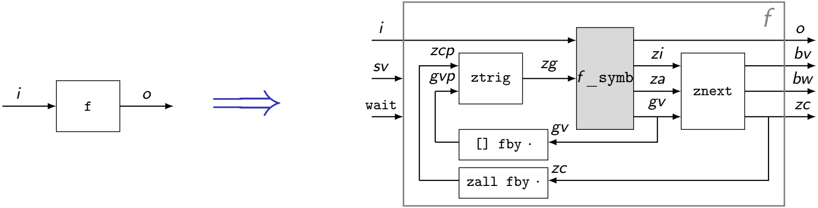
$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} (0,\leq) & (0,\leq) & (-6,\leq) & (-5,<) \\ (20,<) & (0,\leq) & (8,\leq) & (\infty,<) \\ (\infty,<) & (-4,\leq) & (0,\leq) & (\infty,<) \\ (12,\leq) & (\infty,<) & (\infty,<) & (0,\leq) \end{bmatrix} \end{array}$$

- Represents a set of possible clock values.
- Two-dimensional array of *difference constraints*: $t_i - t_j \preceq n$ where $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z} \cup \{\infty\}$.
- One dimension for each clock in the system.
  - row = upper bounds on differences with other clocks.
  - column = lower bounds on differences with other clocks.
- The t_0 clock is always equal to zero (for lower and upper bounds).

DBM interface

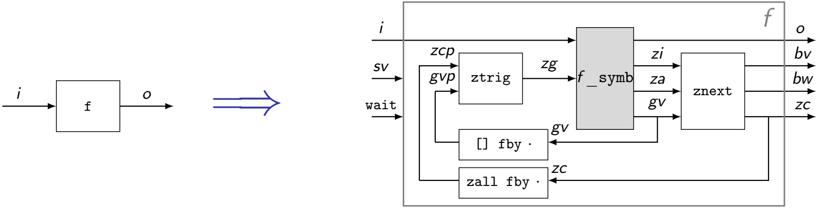
Prototype implemented in OCaml.

- zall The complete space (unconstrained zone).
- zmake(c) Builds a DBM from a single constraint c.
- is_zempty(z) Returns *true* if DBM z denotes an empty zone.
- zreset(z,t,v) Resets a timer t to the value v in zone z.
- zinter(z1, z2) Returns the intersection of zones z1 and z2.
- zinterfold(zv) Returns the intersection of a list of zones zv.
- zup(z) Lets time elapse indefinitely from zone z (drops upper bounds).
- zenabled(zc, gv) Returns a list of booleans characterizing the set of enabled guards in the list gv. A guard is enabled if its activation zone gv_i intersects the current zone zc.
- zdist(zi, g) Returns the activation and deactivation distances of a guard activation zone g from the initial zone zi.
- zdistmap(zi, gv) Returns the list of distances between an initial zone zi and a list of guard activation zones gv.
- zsweep(zi, d1, d2) Sweeps zi between distances d1 and d2.



Source-to-source transformation of hybrid nodes into discrete ones.

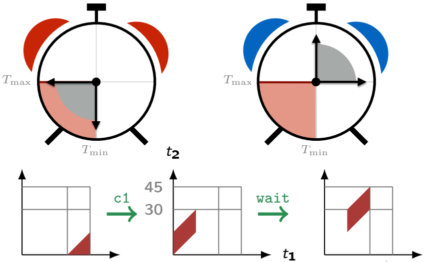
- Replace timers, guards, and invariants.
- Generate a dataflow program manipulating streams of DBMs.

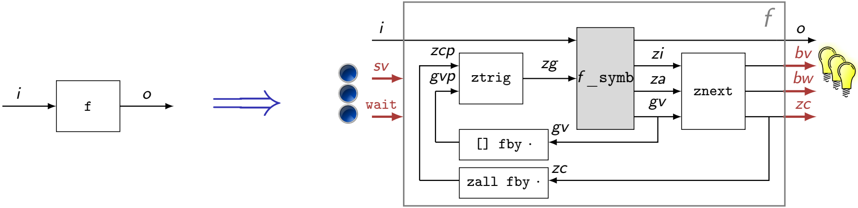


Source-to-source transformation of hybrid nodes into discrete ones.

- Replace timers, guards, and invariants.
- Generate a dataflow program manipulating streams of DBMs.

```
let hybrid clock(t_min, t_max) = c
where
  rec timer t init 0.0 reset c() → 0.0
  and emit c when {t >= t_min}
  and always {t <= t_max}

let hybrid scheduler(t_min, t_max) = c1, c2
where
  rec c1 = clock(t_min, t_max)
  and c2 = clock(t_min, t_max)
```





New inputs

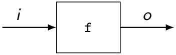
Add 'buttons' that push choice (non-determinism) outside the program.

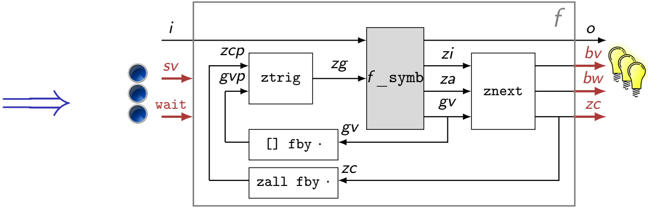
- *sv*: (boolean vector) specifies guards to fire.
- `wait`: (boolean) specifies a wait transition.

New outputs

Add 'light bulbs' that show which buttons are valid.

- *bv*: (boolean vector) indicates enabled guards.
- *bw*: (boolean) indicates that wait is possible.
- *zc*: the current symbolic zone.





New inputs

Add ‘buttons’ that push choice (non-determinism) outside the program.

New outputs

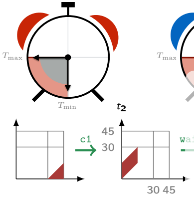
Add ‘light bulbs’ that show which buttons are valid.

let hybrid clock(t_min, t_max) = c

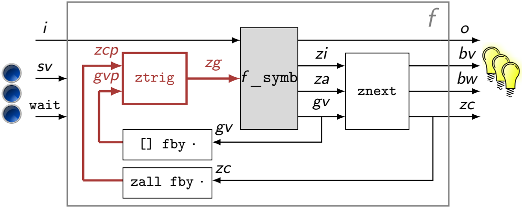
let node clock(wait, c’, (t_min, t_max)) = c, bv, bw, zc

let hybrid scheduler(t_min, t_max) = c1, c2

let node scheduler(wait, (c1’, c2’), (t_min, t_max)) = (c1, c2), bv, bw, zc



bv[0] = true (c1')
bv[1] = false (c2')
bw = false
sv[0] = true (c1')
sv[1] = false (c2')
sw = false

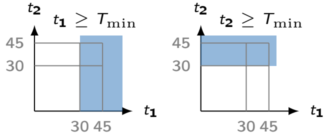


Compute trigger zone of fired guards.

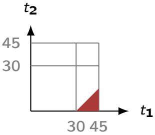
```
let node ztrig(sv, zcp, gvp) = zg where
  rec fv = filter(gvp, sv)
  and zg = zinter(zcp, zinterfold(fv))
```

- Filter enabled guard zones according to user inputs.
- Intersect them with the previous symbolic state.

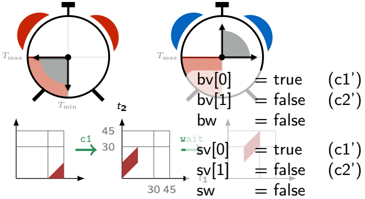
gvp:

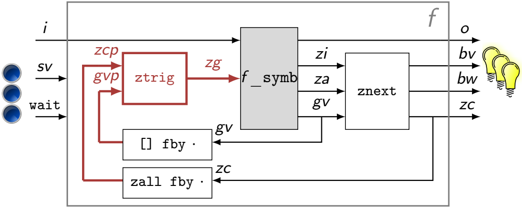


zcp⤳zg:



sv[0] = true sv[1] = false

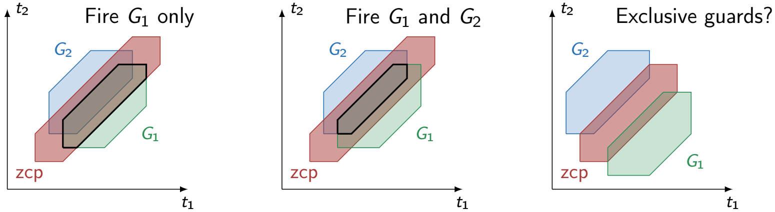


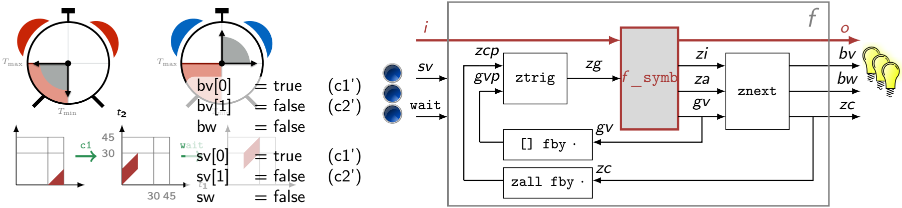


Compute trigger zone of fired guards.

let node ztrig(sv, zcp, gvp) = zg where
  rec fv = filter(gvp, sv)
  and zg = zinter(zcp, zinterfold(fv))

- Filter enabled guard zones according to user inputs.
- Intersect them with the previous symbolic state.



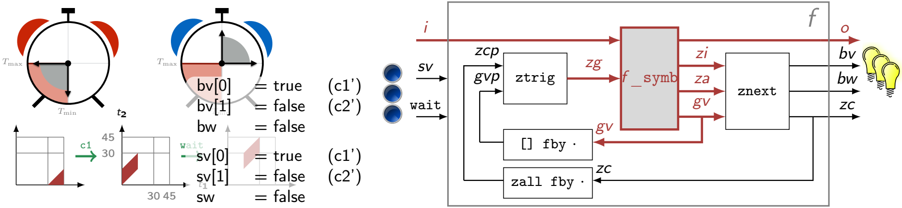


Source-to-source transformation

Defined as 5 mutually recursive functions over syntax.

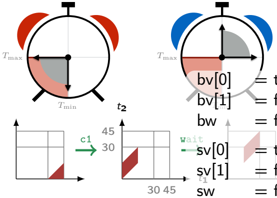
TraDef(d)	translates declarations. Only continuous-time declarations introduced by `hybrid` are modified.
Tra(zi, e)	translates expressions using a variable *zi* to pass the currently computed version of the initial zone.
TraEq(zi, E)	translates equations.
TraZ(zi, c)	translates constraints.
TraH(zi, h)	translates handlers.

Handle discrete computations; implement resets; return updated guards and invariants

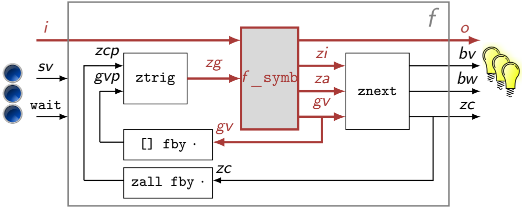



```
let hybrid scheduler(t_min, t_max) = c1, c2 where
  rec c1 = clock(t_min, t_max)
  and c2 = clock(t_min, t_max)
```

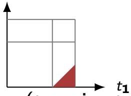
```
let node scheduler_symb((t1, t2), wait, (c1', c2'), zg, (t_min, t_max))
    = (c1, c2), zi, za, gv1 @ gv2 where
  rec c1, zi1, za1, gv1 = clock_symb(t1, wait, c1', zg, (t_min, t_max))
  and c2, zi2, za2, gv2 = clock_symb(t2, wait, c2', zi1, (t_min, t_max))
  and za = zinterfold([za1; za2])
  and zi = if wait then (zall fby zi) else zi2
```



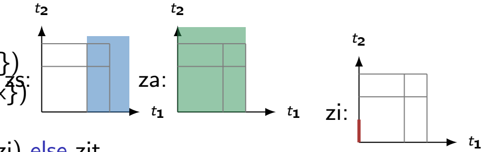

```
bv[0]  = true    (c1')
bv[1]  = false   (c2')
bw     = false
sv[0]  = true    (c1')
sv[1]  = false   (c2')
sw     = false
```

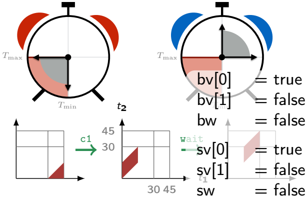



```
let hybrid clock(t_min, t_max) = c where
  rec timer t init 0.0 reset c() → 0.0
  and emit c when {t >= t_min}t2
  and always {t <= t_max}
```

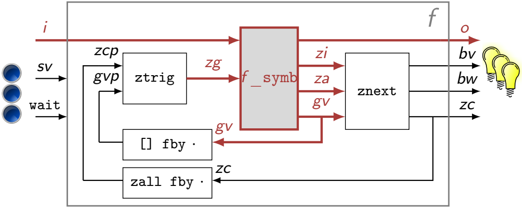



```
                  1  false  true
let node clock_symb(t, wait, c, zg, (t_min, t_max)) = c, zi, za, [zs] where
  rec zit = present (true fby false) → zreset(zg, t, 0)
              | c → zreset(zg, t, 0)
              else zg
  and zs = zmake({t >= t_min})
  and zb = zmake({t <= t_max})
  and za = zinterfold([zb])
  and zi = if wait then (zall fby zi) else zit
```

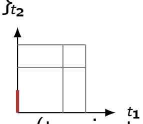




bv[0]	= true	(c1')
bv[1]	= false	(c2')
bw	= false	
sv[0]	= true	(c1')
sv[1]	= false	(c2')
sw	= false	



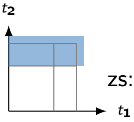

```
let hybrid clock(t_min, t_max) = c where
  rec timer t init 0.0 reset c() → 0.0
  and emit c when {t >= t_min}
  and always {t <= t_max}
```



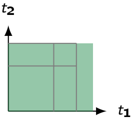
2 false false

```
let node clock_symb(t, wait, c, zg, (t_min, t_max)) = c, zi, za, [zs] where
  rec zit = present (true fby false) → zreset(zg, t, 0)
          | c → zreset(zg, t, 0)
          else zg
  and zs = zmake({t >= t_min})
  and zb = zmake({t <= t_max})
  and za = zinterfold([zb])
  and zi = if wait then (zall fby zi) else zit
```

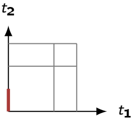
za:

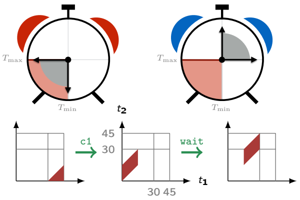


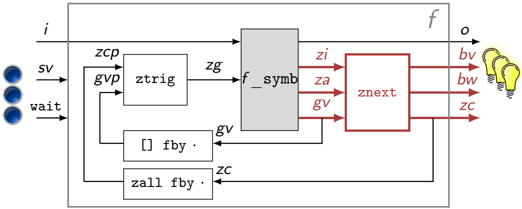
zs:



zi:



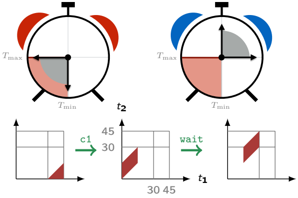


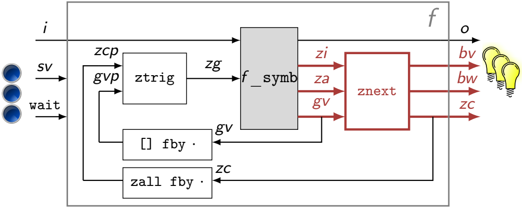


Compute *next* symbolic state and enabled transitions

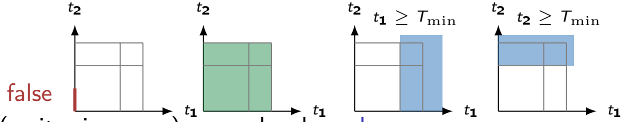
- Take initial zone zi, invariant conjunction za, and guard zone vector gv.
- Compute the symbolic state and the transition ‘lights’.

```
let node znext(wait, zi, za, gv) = zc, bv, bw where
  rec dp = if wait then (dzero fby d) else dzero
  and dl = zdistmap(zi, gv)
  and d  = mindist(dl, dp)
  and zn = zsweep(zi, dp, d)
  and zc = zinter(zn, za)
  and bv = zenabled(zc, gv)
  and zm = zinter(zup(zn), za)
  and bw = (zc <> zm)
```



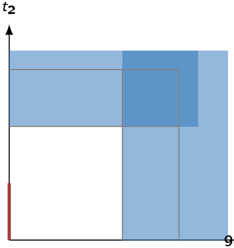


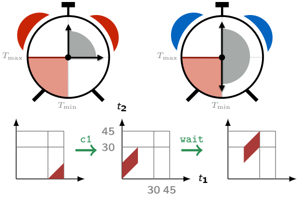
Compute *next* symbolic state and enabled transitions

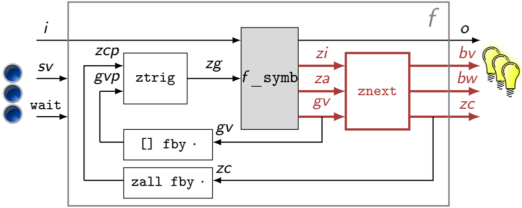



```
let node znext(wait, zi, za, gv) = zc, bv, bw where
  rec dp = if wait then (dzero fby d) else dzero
  and dl = zdistmap(zi, gv)
  and d  = mindist(dl, dp)
  and zn = zsweep(zi, dp, d)
  and zc = zinter(zn, za)
  and bv = zenabled(zc, gv)
  and zm = zinter(zup(zn), za)
  and bw = (zc <> zm)
```

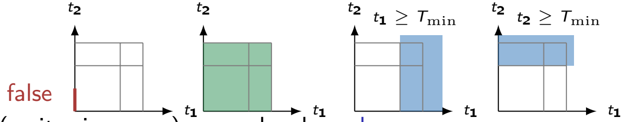
$$
\begin{aligned}
dp &= (0\ , \leq) \\
dl &= (15, \leq) \\
 & \quad (30, \leq) \\
 & \quad (\infty, \leq) \\
d &= (15, \leq)
\end{aligned}
$$





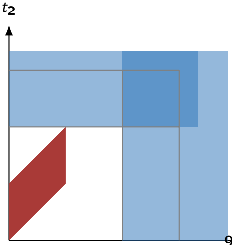


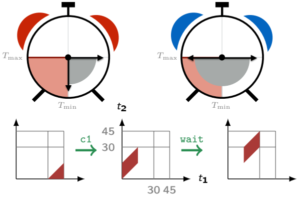
Compute *next* symbolic state and enabled transitions

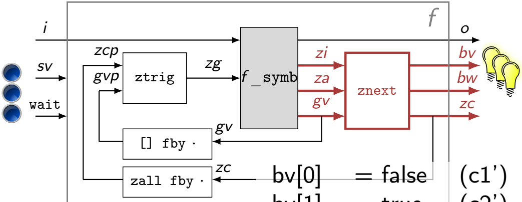



```
let node znext(wait, zi, za, gv) = zc, bv, bw where
  rec dp = if wait then (dzero fby d) else dzero
  and dl = zdistmap(zi, gv)
  and d  = mindist(dl, dp)
  and zn = zsweep(zi, dp, d)
  and zc = zinter(zn, za)
  and bv = zenabled(zc, gv)
  and zm = zinter(zup(zn), za)
  and bw = (zc <> zm)
```

$$
\begin{aligned}
dp &= (0\ , \leq) \\
dl &= (15, \leq) \\
& \quad\ (30, \leq) \\
& \quad\ (\infty, \leq) \\
d &= (15, \leq)
\end{aligned}
$$



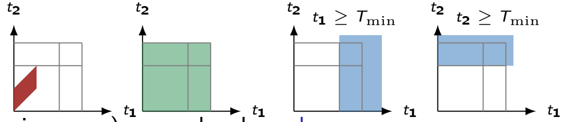




bv[0] = false (c1')
bv[1] = true (c2')
bw = false

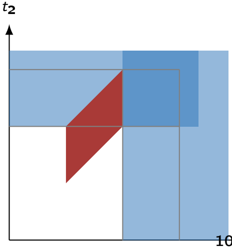
Compute *next* symbolic state and enabled transitions

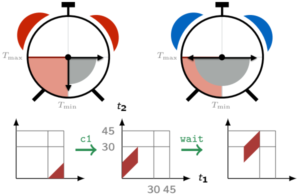
true

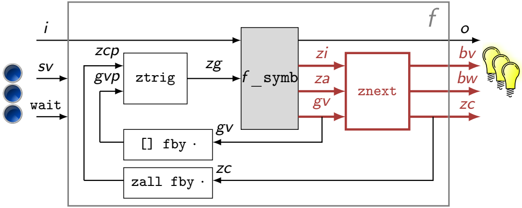



```
let node znext(wait, zi, za, gv) = zc, bv, bw where
  rec dp = if wait then (dzero fby d) else dzero
  and dl = zdistmap(zi, gv)
  and d  = mindist(dl, dp)
  and zn = zsweep(zi, dp, d)
  and zc = zinter(zn, za)
  and bv = zenabled(zc, gv)
  and zm = zinter(zup(zn), za)
  and bw = (zc <> zm)
```

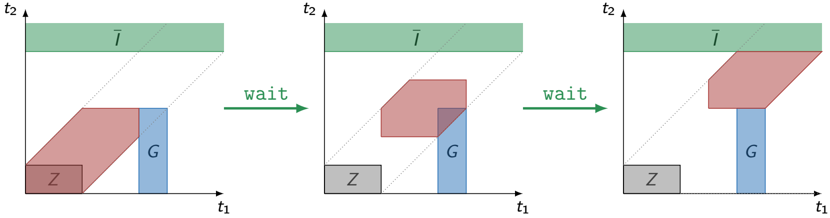
$$
\begin{aligned}
\mathrm{dp} &= (15, \leq) \\
\mathrm{dl} &= (15, \leq) \\
& (\infty, \leq) \\
\mathrm{d} &= (15, \leq)
\end{aligned}
$$

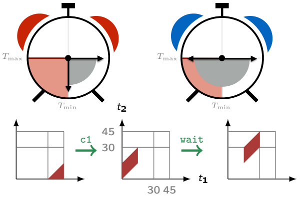


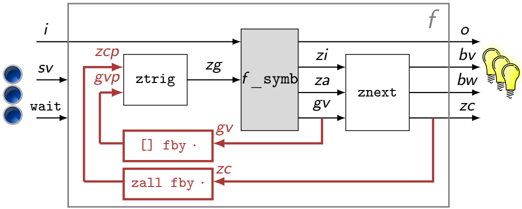




Compute *next* symbolic state and enabled transitions

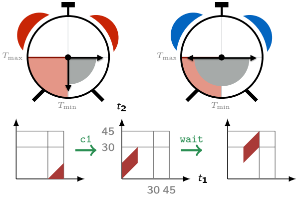


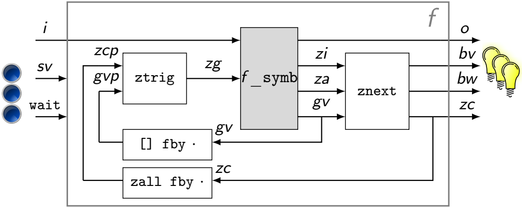




Feedback (fby) is critical

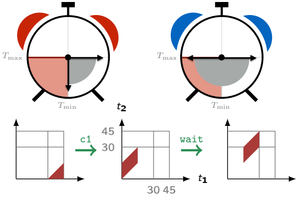
- Avoid multiple passes by calculating in one cycle and using in the next.
- Remember the next active guard zones.
- Remember the next active symbolic state.

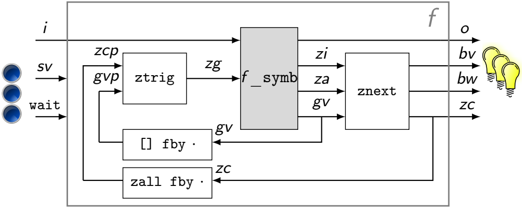




Express compositions and delays in discrete subset of language

```
let node clock(wait, c, (t_min, t_max)) = c', bv, bw, zc where
  rec zg = ztrig([c], zcp, gvp)
  and c', zi, za, gv = clock_symb(1, wait, c, zg, (t_min, t_max))
  and zc, bv, bw = znext(wait, zi, za, gv)
  and zcp = zall fby zc
  and gvp = [] fby gv

let node scheduler(wait, (c1', c2'), (t_min, t_max))
    = (c1, c2), bv, bw, zc where
  rec zg = ztrig([c1'; c2'], zcp, gvp)
  and (c1, c2), zi, za, gv =
    scheduler_symb((1, 2), wait, (c1', c2'), zg, (t_min, t_max))
  and zc, bv, bw = znext(wait, zi, za, gv)
  and zcp = zall fby zc
```





Summary of 3 execution phases

1. From current zone zcp and vector of guard activation zones gvp (from previous step), `ztrig` computes the trigger zone zg.

2. f `_symb` triggers discrete-time computations and returns zi obtained by applying resets to zg, the conjunction of active invariants za, and the new vector of guard zones gv.

3. `znext` computes the new zone zc by letting time elapse from zi until the set of enabled guards changes.

Comparison: Uppaal vs Zsy

Uppaal

- First-rate graphical interface and simulator.
- Verification by model-checking.
- Highly-optimized DBM library.
- Single-level of parallel composition of instantiated templates.
- C-like language for combinatorial functions.
- Sophisticated semantics implemented inside tool.

Zsy

- Hierarchical parallel compositions.
- Lustre-like language for stateful functions.
- Semantics encoded by source-to-source transformation.

Zsy: summary and future work

Summary

- A novel Lustre-like language with Timed Automaton features. [Baudart, Bourke, and Pouzet (2017): Symbolic Simulation of Dataflow Synchronous Programs with Timers]
- Source-to-source compilation schema for symbolic simulations.
- Novel 'sweeping' construct for explicit `wait` transitions
- Prototype implementation: https://github.com/gbdrt/zsy/tree/fdl17

Future directions

- Generate C and link with Uppaal DBM library?
- Incorporate richer domains? [Miné (2006): The octagon abstract domain]
- Implement support for state machines? [Baudart (2017): A Synchronous Approach to Quasi-Periodic Systems]
- Verification by symbolic model-checking?

Outline

Summary

- The Quasi-Periodic Architecture (Synchronous Real-Time Model) arises naturally whenever two or more sampling controllers are interconnected; it is widely used.

- Synchronous languages can be used to model and verify such systems but care is required when abstracting from real-time details.

- Discrete controller logic can be specified as a synchronous program and implemented on a distributed architecture using, for instance,
  - Clock synchronization
  - Back-pressure flow control
  - A simple real-time protocol

- Lustre + Timed Automaton features can be compiled into Lustre on flows of DBMs to give symbolic simulations.

References I

- Alur, R. and D. L. Dill (Apr. 1994). "A Theory of Timed Automata". In: *Theor. Comp. Sci.* 126.2, pp. 183–235.
- Baudart, G. (Mar. 2017). "A Synchronous Approach to Quasi-Periodic Systems". PhD thesis. PSL Research University.
- Baudart, G., T. Bourke, and M. Pouzet (Oct. 2016). "Soundness of the Quasi-Synchronous Abstraction". In: *Proc. 16th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2016)*. Ed. by R. Piskac and M. Talupur. IEEE. Mountain View, CA, USA, pp. 9–16.
- — (Sept. 2017). "Symbolic Simulation of Dataflow Synchronous Programs with Timers". In: *Proc. 12th Forum on Specification and Design Languages (FDL 2017)*. Verona, Italy.
- Behrmann, G., A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks (Sept. 2006). "Uppaal 4.0". In: *3rd Int. Conf. on Quantitative Evaluation of Systems*. IEEE Computer Society. Riverside, California, USA, pp. 125–126.

References II

- Benveniste, A., P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis (Oct. 2002). "A Protocol for Loosely Time-Triggered Architectures". In: *Proc. 2nd Int. Conf. on Embedded Software (EMSOFT 2002)*. Ed. by A. L. Sangiovanni-Vincentelli and J. Sifakis. Vol. 2491. LNCS. Grenoble, France: Springer, pp. 252–265.
- Berry, G. (Aug. 1989). "Real Time Programming: Special Purpose or General Purpose Languages". In: *Proc. 11th Int. Federation for Information Processing (IFIP) World Computer Congress*. Ed. by G. Ritter. San Francisco, USA, pp. 11–17.
- Bourke, T. and M. Pouzet (Apr. 2013). "Zélus: A Synchronous Language with ODEs". In: *Proc. 16th Int. Conf. on Hybrid Systems: Computation and Control (HSCC 2013)*. Ed. by C. Belta and F. Ivancic. Philadelphia, USA: ACM Press, pp. 113–118.
- Caspi, P., D. Pilaud, N. Halbwachs, and J. Plaice (Jan. 1987). "LUSTRE: A declarative language for programming synchronous systems". In: *Proc. 14th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1987)*. Munich, Germany: ACM Press, pp. 178–188.

References III

- Caspi, P. (May 2000). *The Quasi-Synchronous Approach to Distributed Control Systems*. Tech. rep. CMA/009931. VERIMAG, Crysis Project.
- — (Oct. 2001). "Embedded Control: From Asynchrony to Synchrony and Back". In: *Proc. 1st Int. Conf. on Embedded Software (EMSOFT 2001)*. Ed. by T. A. Henzinger and C. M. Kirsch. Vol. 2211. LNCS. Tahoe City, USA: Springer, pp. 80–99.
- Dill, D. L. (1989). "Timing assumptions and verification of finite-state concurrent systems". In: *Proc. Int. Workshop on Automatic Verification Methods for Finite State Systems*. LNCS 407. Grenoble, France: Springer, pp. 197–212.
- Hagen, G. and C. Tinelli (Nov. 2008). "Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques". In: *Proc. 8th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2008)*. Ed. by A. Cimatti and R. B. Jones. IEEE. Portland, OR, USA, Article 15.

References IV

- Halbwachs, N. and S. Baghdadi (Oct. 2002). "Synchronous modelling of asynchronous systems". In: *Proc. 2nd Int. Conf. on Embedded Software (EMSOFT 2002)*. Ed. by A. L. Sangiovanni-Vincentelli and J. Sifakis. Vol. 2491. LNCS. Grenoble, France: Springer, pp. 240–251.
- Halbwachs, N., F. Lagnier, and C. Ratel (Sept. 1992). "Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE". In: *IEEE Trans. Software Engineering* 18.9, pp. 785–793.
- Halbwachs, N. and L. Mandel (June 2006). "Simulation and Verification of Aysnchronous Systems by means of a Synchronous Model". In: *Proc. 6th Int. Conf. on Application of Concurrency to System Design (ACSD 2006)*. IEEE Computer Society. Turku, Finland: IEEE, pp. 3–14.
- Henzinger, T. A., X. Nicollin, J. Sifakis, and S. Yovine (June 1994). "Symbolic Model Checking for Real-Time Systems". In: *Information and Computation* 111.2, pp. 192–244.
- Hergé (1955). *Les Cigares du Pharaon*. Tintin. Casterman.

References V

- Isenberg, T. and H. Wehrheim (Nov. 2014). "Timed Automata Verification via IC3 with Zones". In: *16th Int. Conf. on Formal Methods and Software Engineering (ICFEM 2014)*. Ed. by S. Merz and J. Pang. Vol. 8829. LNCS. Luxembourg: Springer, pp. 203–218.
- Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers.
- Kopetz, H. (2011). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd ed. Kluwer Academic Publishers.
- Kopetz, H. and G. Bauer (Jan. 2003). "The Time-Triggered Architecture". In: *Proc. IEEE* 91.1, pp. 112–126.
- Lamport, L. (July 1977). "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Comms. ACM* 21.7, pp. 558–565.
- Miller, S. P., S. Bhattacharyya, C. Tinelli, S. Smolka, C. Sticksel, B. Meng, and J. Yang (2015). *Formal Verification of Quasi-Synchronous Systems*. Tech. rep. DTIC Document.

References VI

- Milner, R. (1989). *Communication and Concurrency*. Upper Saddle River, NJ: Prentice-Hall, Inc.
- Miné, A. (2006). "The octagon abstract domain". In: *Higher-Order and Symbolic Computation* 19.1, pp. 31–100.
- Neumann, P. G. (1994). *Computer Related Risks*. Addison Wesley.
- Sangiovanni-Vincentelli, A. L. and J. Sifakis, eds. (Oct. 2002). *Proc. 2nd Int. Conf. on Embedded Software (EMSOFT 2002)*. Vol. 2491. LNCS. Grenoble, France: Springer.
- Vaandrager, F. and A. de Groot (Dec. 2006). "Analysis of a Biphase Mark Protocol with Uppaal and PVS". In: *Formal Aspects of Computing* 18.4, pp. 433–458.