A Synchronous Approach to Quasi-Periodic Systems

PhD Defense — Guillaume Baudart

March 13, 2017

Embedded Systems

Embedded Systems

Reactive systems:

- constant interaction with the environment
- for an unbounded amount of time
- must not fail

Embedded Systems

Reactive systems:

- constant interaction with the environment
- for an unbounded amount of time
- must not fail

Quasi-periodic systems:

- several computing nodes
- unsynchronized architecture

Embedded Systems

Reactive systems:

- constant interaction with the environment
- for an unbounded amount of time
- must not fail

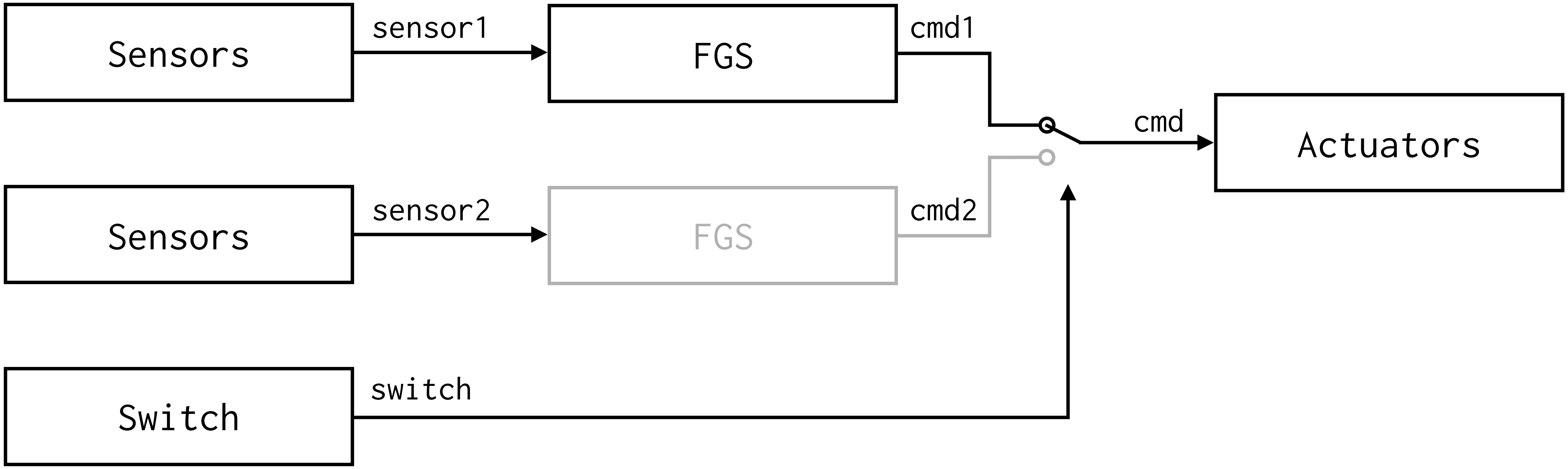
Quasi-periodic systems:

- several computing nodes
- unsynchronized architecture

aircraft, nuclear plants, trains, cars...

Quasi-Periodic Systems

Example: Flight Control System



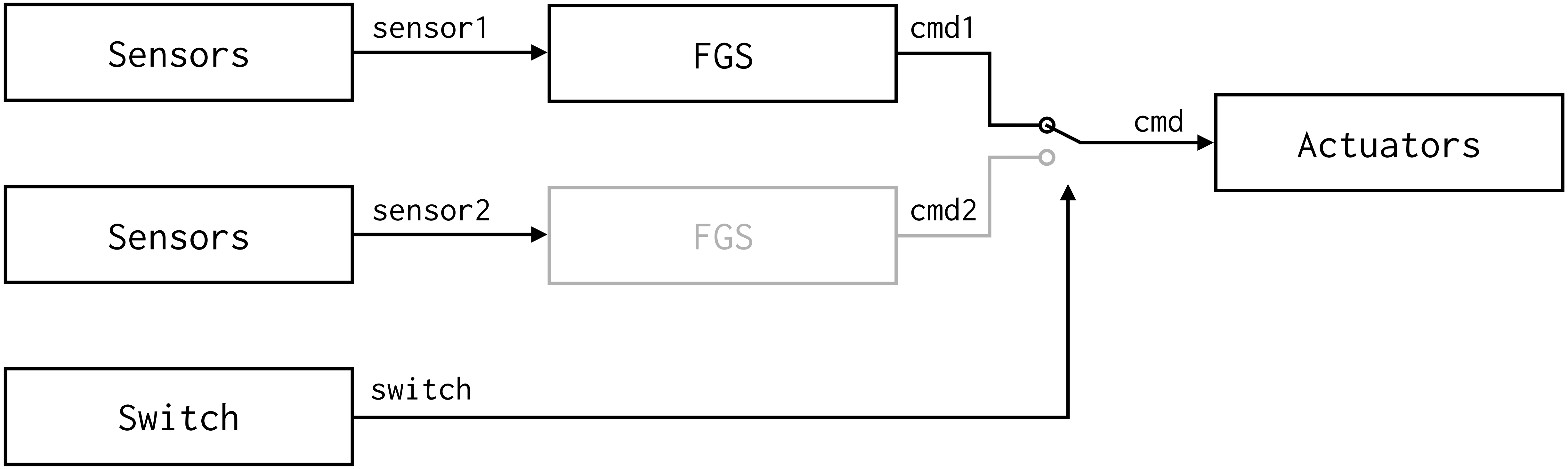
Generate pitch and roll guidance commands

Example from [MBT+15]

Quasi-Periodic Systems

Example: Flight Control System

Two redundant Flight Guidance Systems
Only one active side (pilot side)



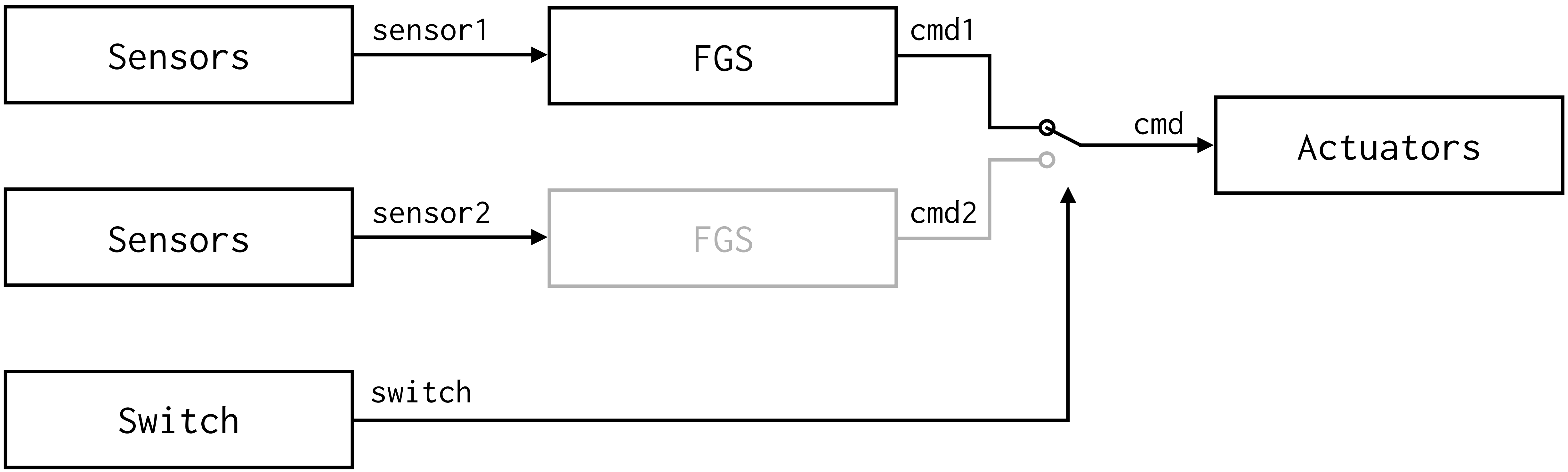
Generate pitch and roll guidance commands

Example from [MBT+15]

Quasi-Periodic Systems

Example: Flight Control System

Two redundant Flight Guidance Systems
Only one active side (pilot side)



Crew can switch from one to the other

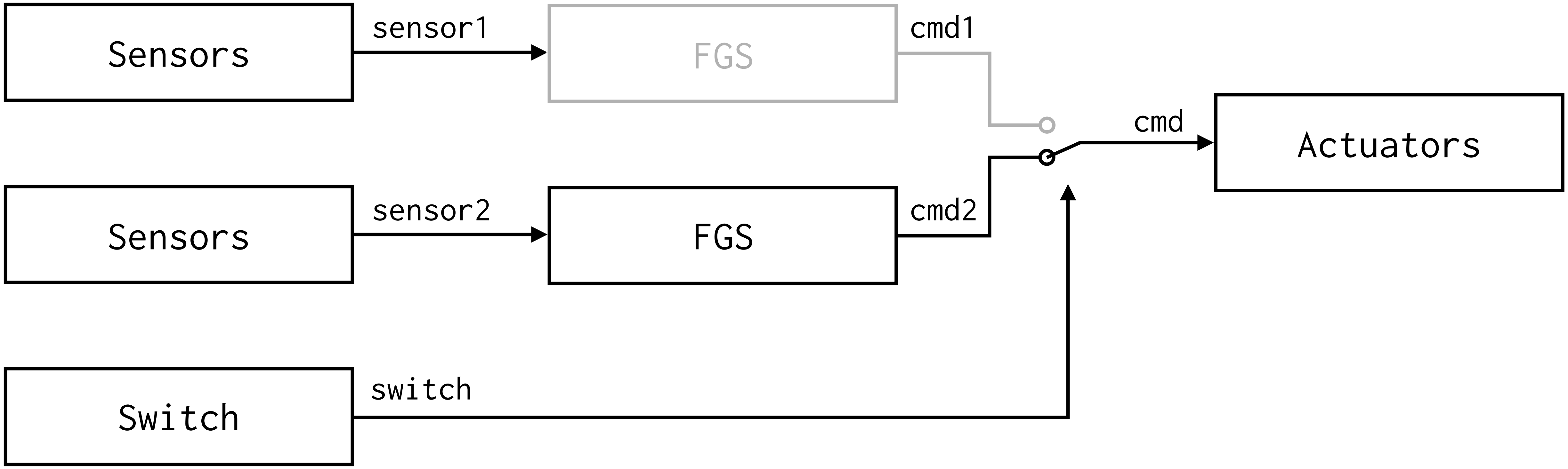
Generate pitch and roll guidance commands

Example from [MBT+15]

Quasi-Periodic Systems

Example: Flight Control System

Two redundant Flight Guidance Systems
Only one active side (pilot side)



Crew can switch from one to the other

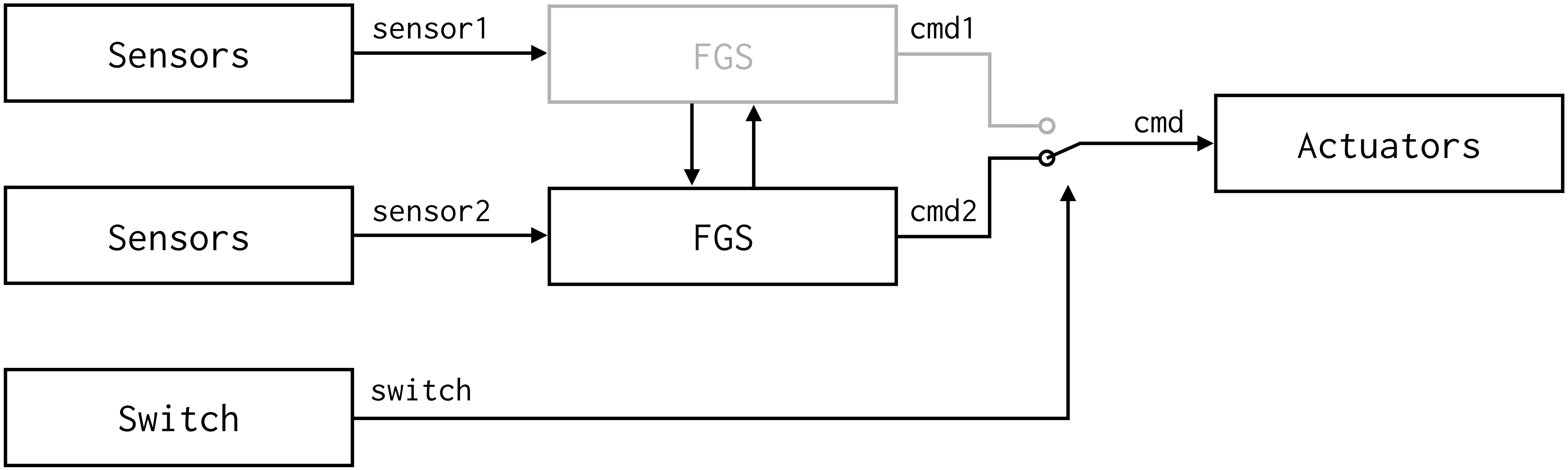
Generate pitch and roll guidance commands

Example from [MBT+15]

Quasi-Periodic Systems

Example: Flight Control System

Two redundant Flight Guidance Systems
Only one active side (pilot side)



Crew can switch from one to the other

The two modules must share information to avoid control glitch

Generate pitch and roll guidance commands

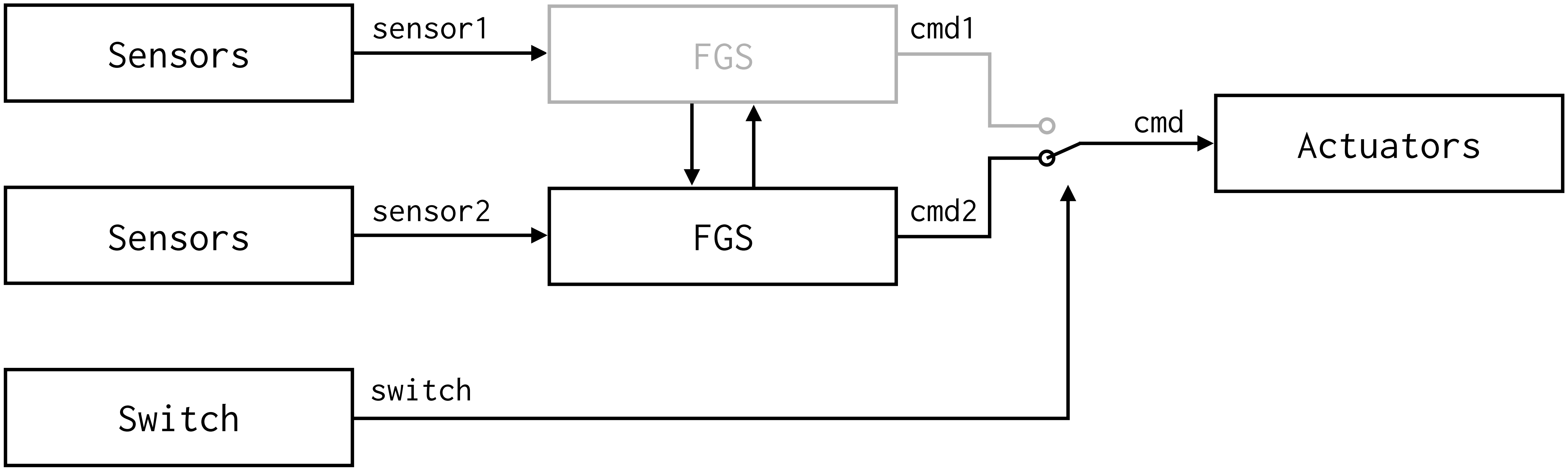
Example from [MBT+15]

Quasi-Periodic Systems

Example: Flight Control System

Two redundant Flight Guidance Systems
Only one active side (pilot side)

Run embedded application...



Crew can switch from one to the other

The two modules must share information to avoid control glitch

Generate pitch and roll guidance commands

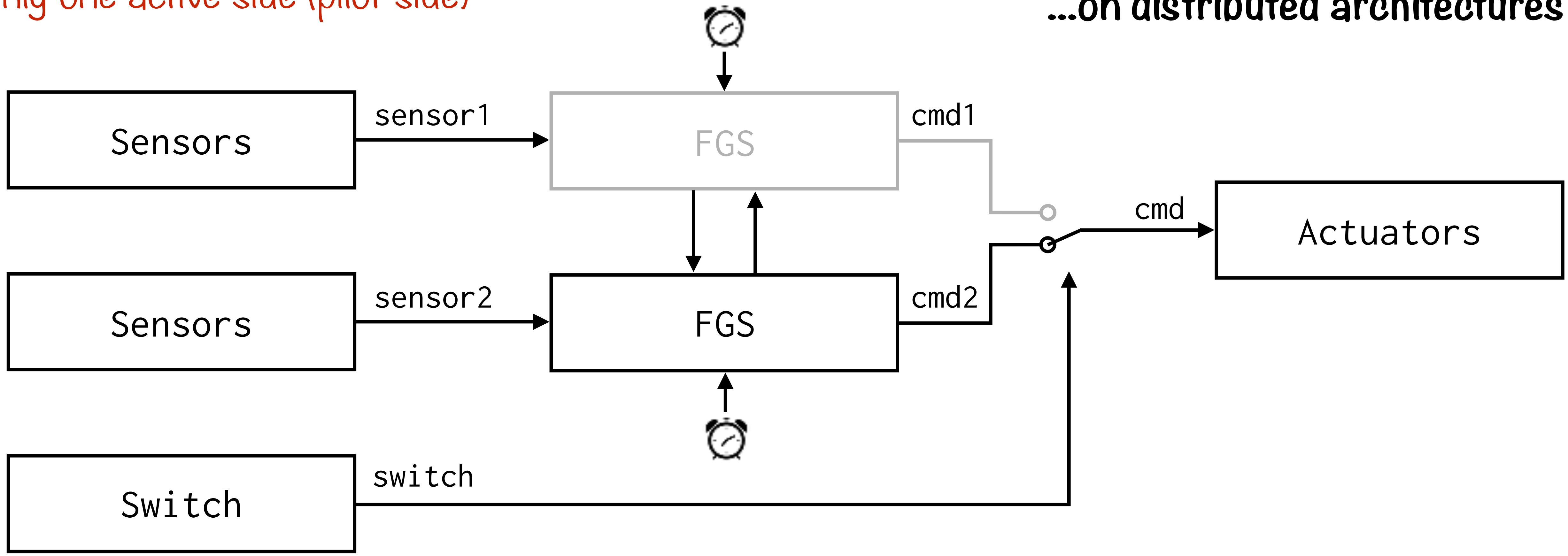
Example from [MBT+15]

Quasi-Periodic Systems

Example: Flight Control System

Two redundant Flight Guidance Systems
Only one active side (pilot side)

Run embedded application...
...on distributed architectures



Crew can switch from one to the other

The two modules must share information to avoid control glitch

Generate pitch and roll guidance commands

Example from [MBT+15]

Quasi-Periodic Architecture

For each process, activations are triggered by a **local clock**
Execution: infinite sequence of activations

- For each process: **known bounds** for the time between two activations

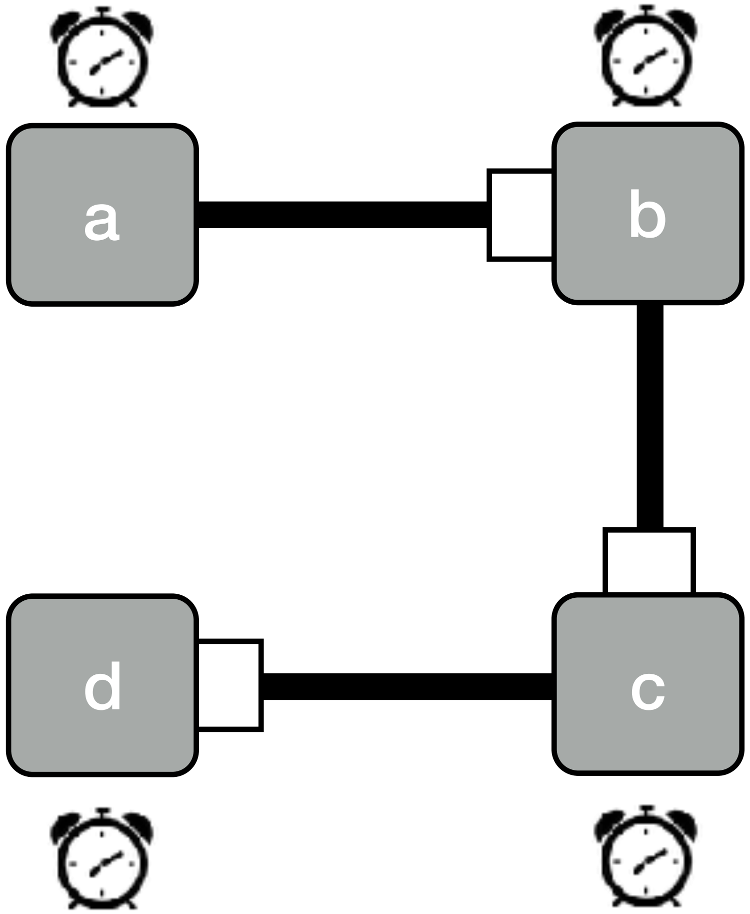
  $0 \leq T_{\min} \leq \kappa_{i+1} - \kappa_i \leq T_{\max}$

  $(\kappa_i)_{i \in \mathbb{N}}$ clock activations

- **Buffered communication** without message inversion or loss

- **Bounded communication** delay

  $0 \leq \tau_{\min} \leq \tau \leq \tau_{\max}$



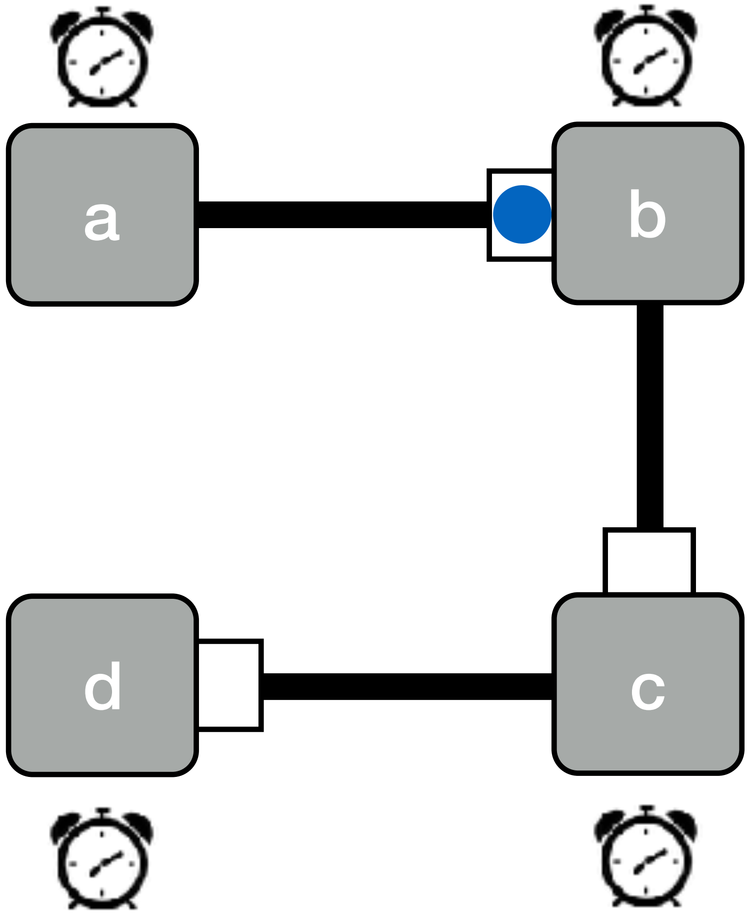
Quasi-Periodic Architecture

For each process, activations are triggered by a **local clock**
Execution: infinite sequence of activations

- For each process: **known bounds** for the time between two activations

$$0 \leq T_{\min} \leq \kappa_{i+1} - \kappa_i \leq T_{\max}$$

$(\kappa_i)_{i \in \mathbb{N}}$ clock activations



- **Buffered communication** without message inversion or loss

- **Bounded communication** delay

$$0 \leq \tau_{\min} \leq \tau \leq \tau_{\max}$$

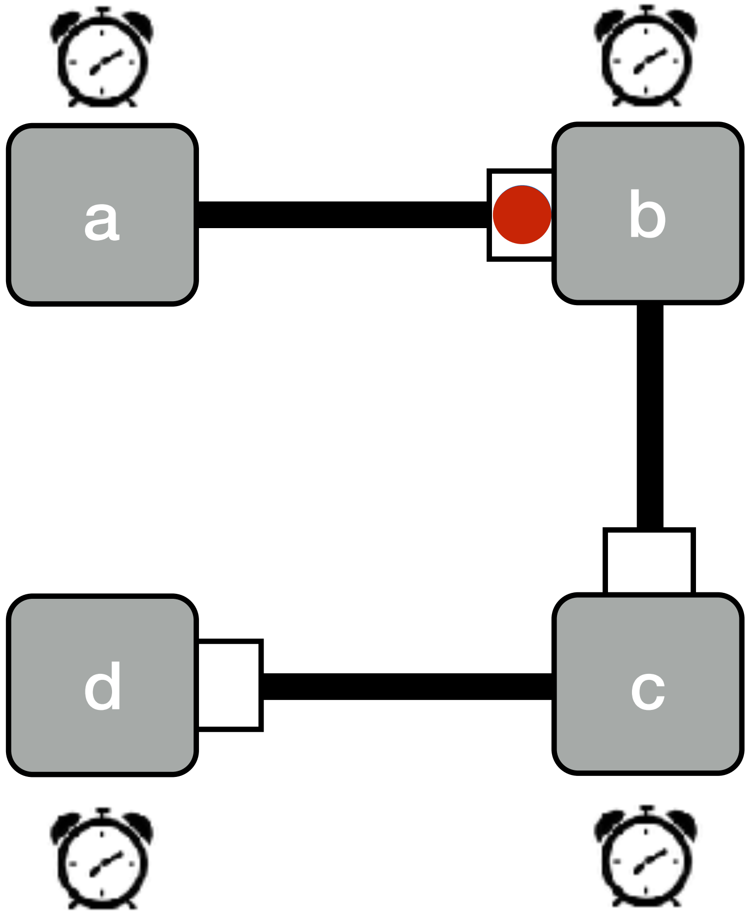
Quasi-Periodic Architecture

For each process, activations are triggered by a **local clock**
Execution: infinite sequence of activations

- For each process: **known bounds** for the time between two activations

$$0 \leq T_{\min} \leq \kappa_{i+1} - \kappa_i \leq T_{\max}$$

$(\kappa_i)_{i \in \mathbb{N}}$ clock activations



- **Buffered communication** without message inversion or loss

- **Bounded communication** delay

$$0 \leq \tau_{\min} \leq \tau \leq \tau_{\max}$$

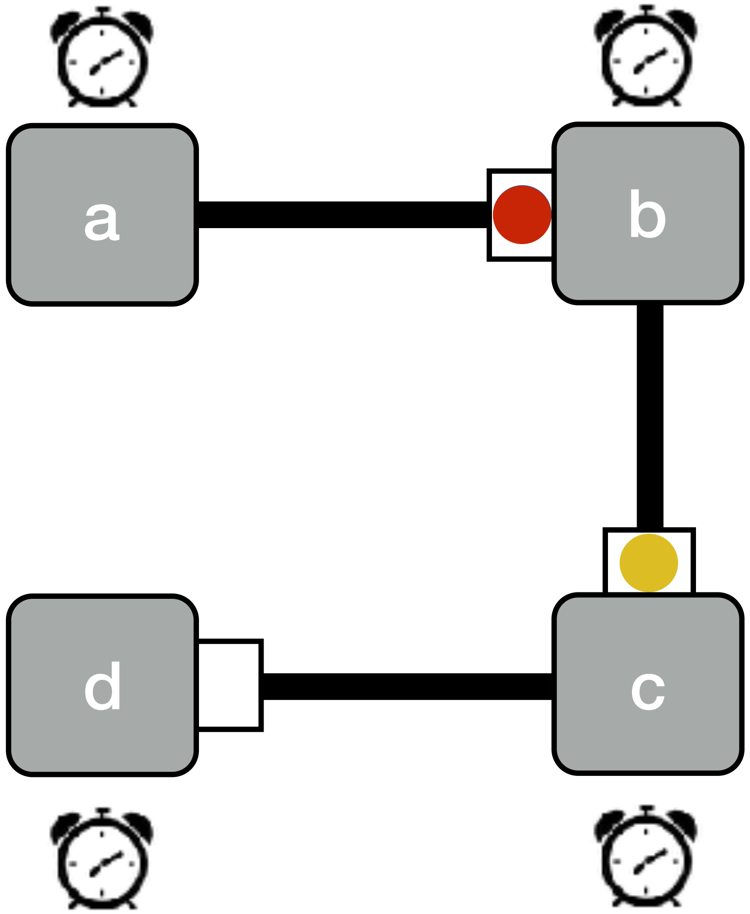
Quasi-Periodic Architecture

For each process, activations are triggered by a **local clock**
Execution: infinite sequence of activations

- For each process: **known bounds** for the time between two activations

$$0 \leq T_{\min} \leq \kappa_{i+1} - \kappa_i \leq T_{\max}$$

$(\kappa_i)_{i \in \mathbb{N}}$ clock activations



- **Buffered communication** without message inversion or loss

- **Bounded communication** delay

$$0 \leq \tau_{\min} \leq \tau \leq \tau_{\max}$$

Quasi-Periodic Architecture

For each process, activations are triggered by a **local clock**
Execution: infinite sequence of activations

- For each process: **known bounds** for the time between two activations

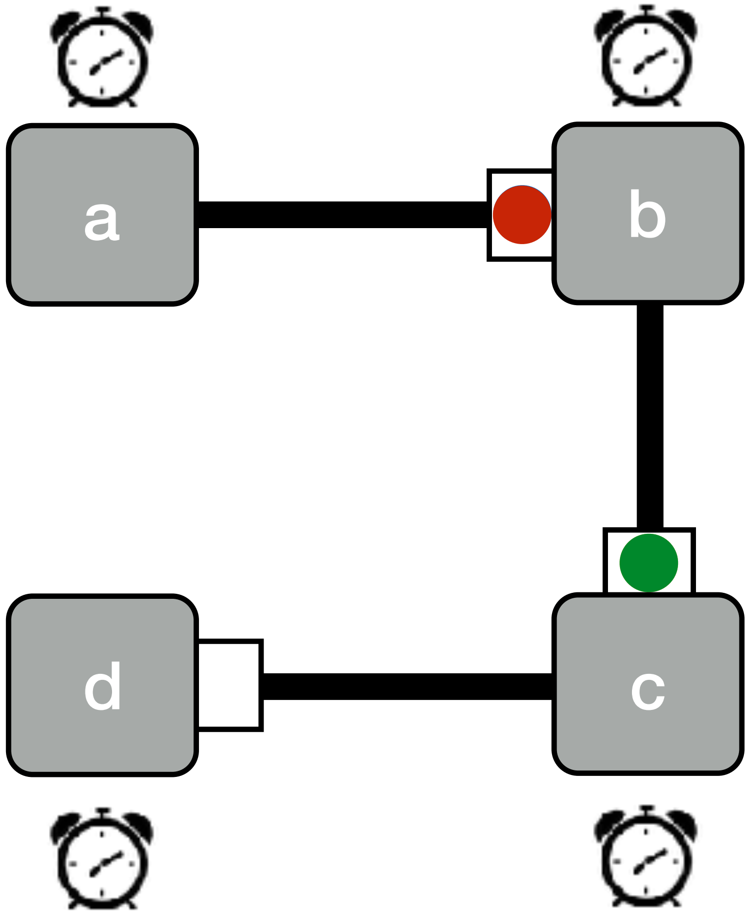
$$0 \leq T_{\min} \leq \kappa_{i+1} - \kappa_i \leq T_{\max}$$

$(\kappa_i)_{i \in \mathbb{N}}$ clock activations

- **Buffered communication** without message inversion or loss

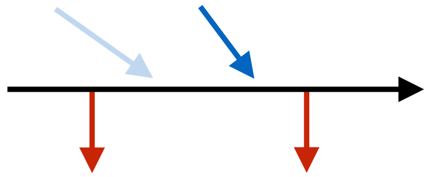
- **Bounded communication** delay

$$0 \leq \tau_{\min} \leq \tau \leq \tau_{\max}$$

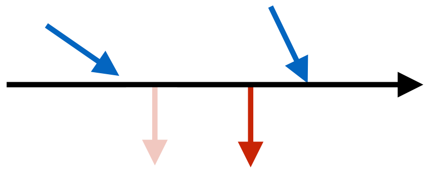


Sampling Artifacts

- **Overwriting:** loss of values

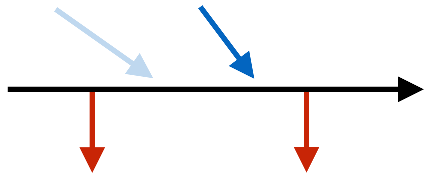


- **Oversampling:** duplication of values

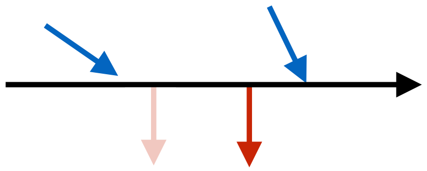


Sampling Artifacts

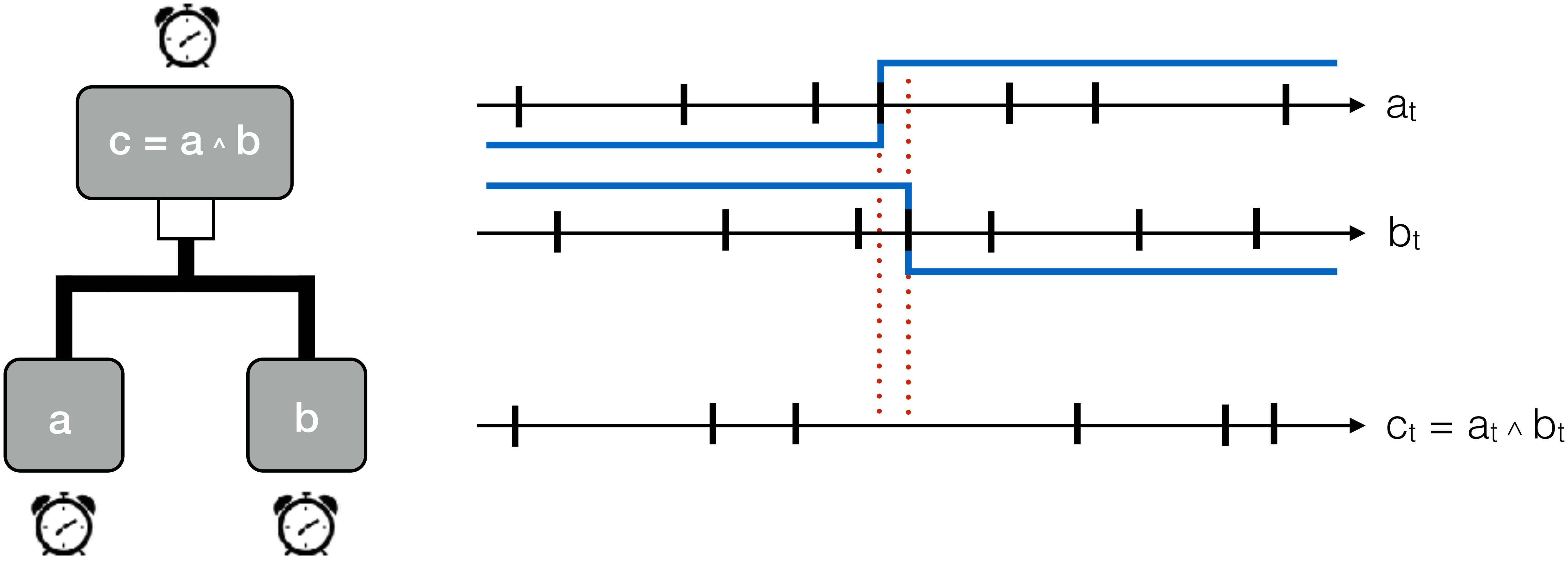
- **Overwriting:** loss of values



- **Oversampling:** duplication of values



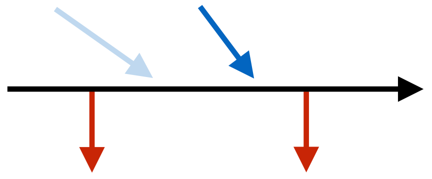
- **Combination of signals**



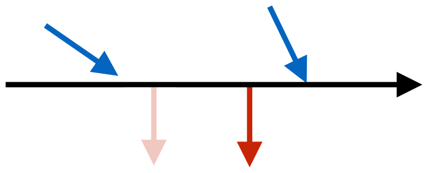
Example from [Cas00]

Sampling Artifacts

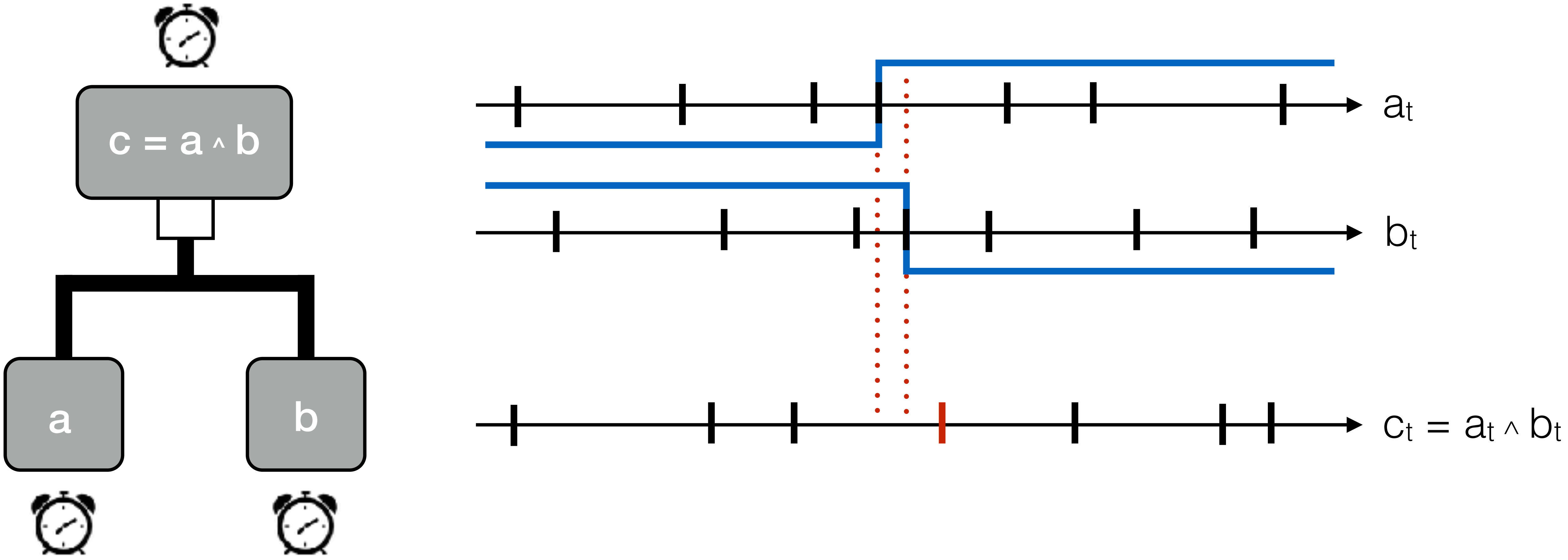
- **Overwriting:** loss of values



- **Oversampling:** duplication of values



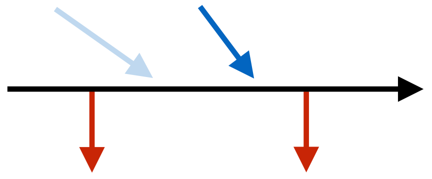
- **Combination of signals**



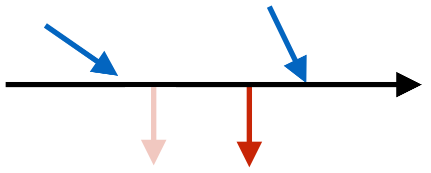
Example from [Cas00]

Sampling Artifacts

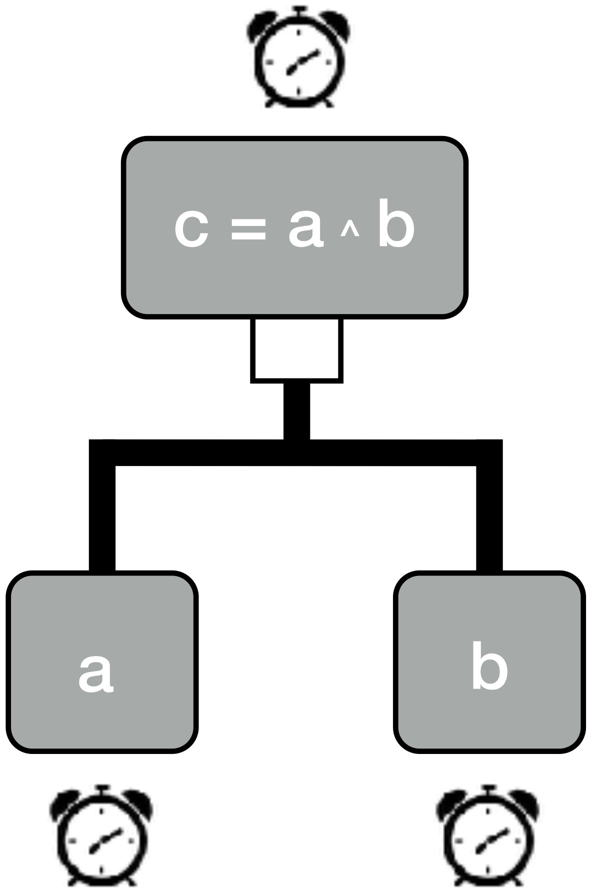
- **Overwriting:** loss of values

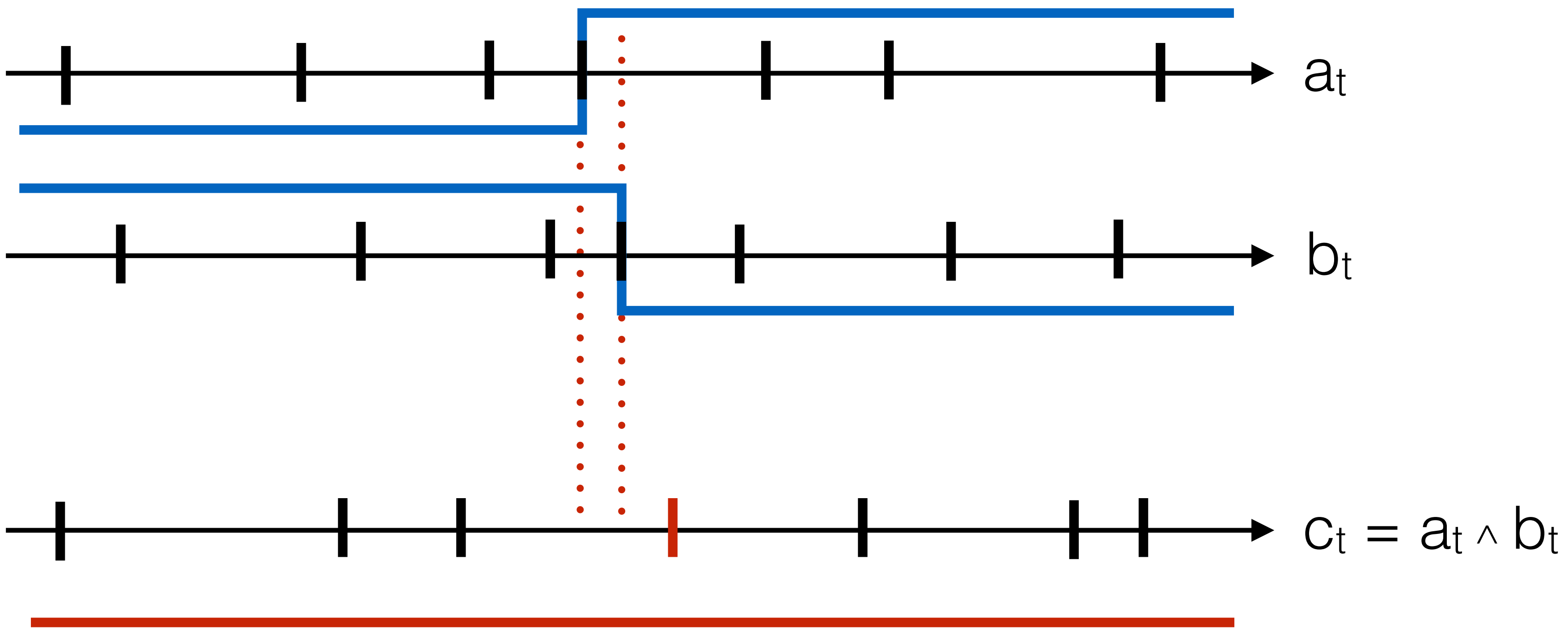


- **Oversampling:** duplication of values



- **Combination of signals**

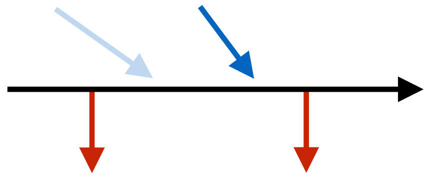




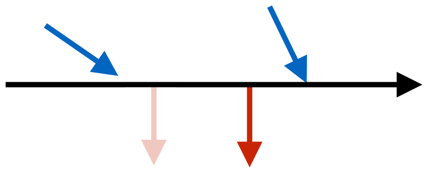
Example from [Cas00]

Sampling Artifacts

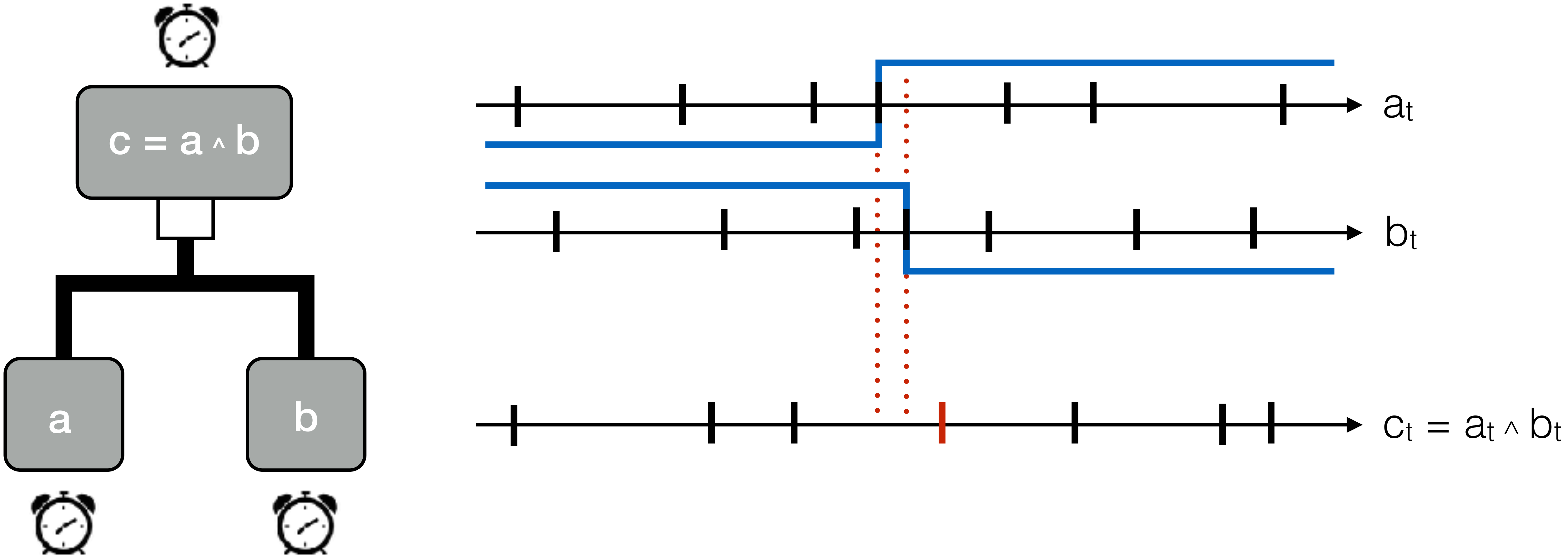
- **Overwriting:** loss of values



- **Oversampling:** duplication of values



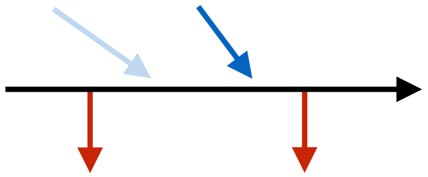
- **Combination of signals**



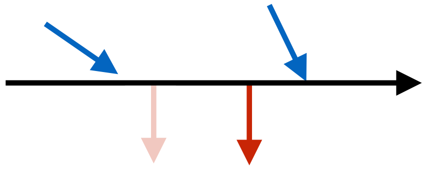
Example from [Cas00]

Sampling Artifacts

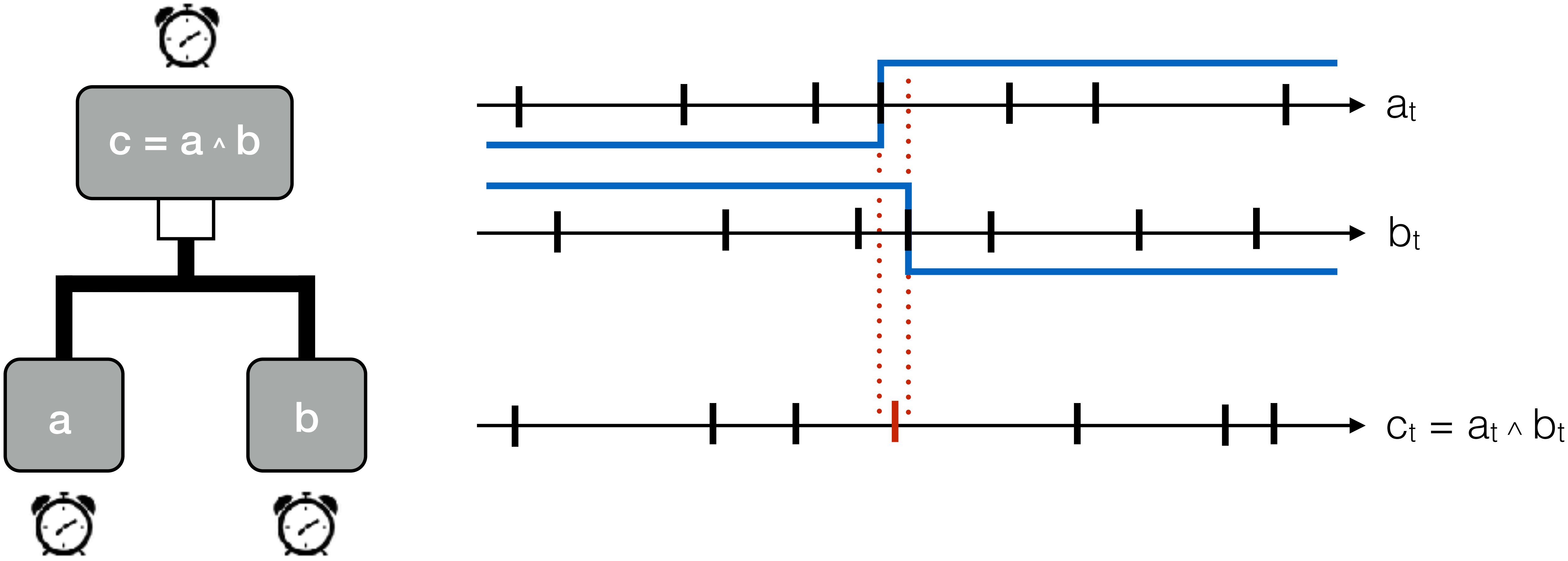
- **Overwriting:** loss of values



- **Oversampling:** duplication of values



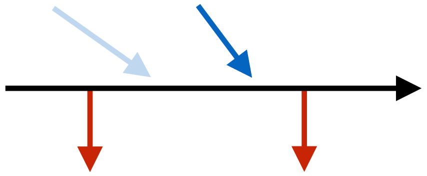
- **Combination of signals**



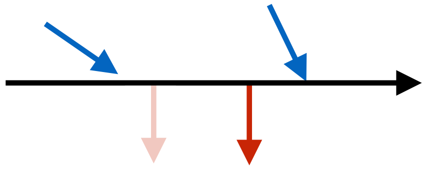
Example from [Cas00]

Sampling Artifacts

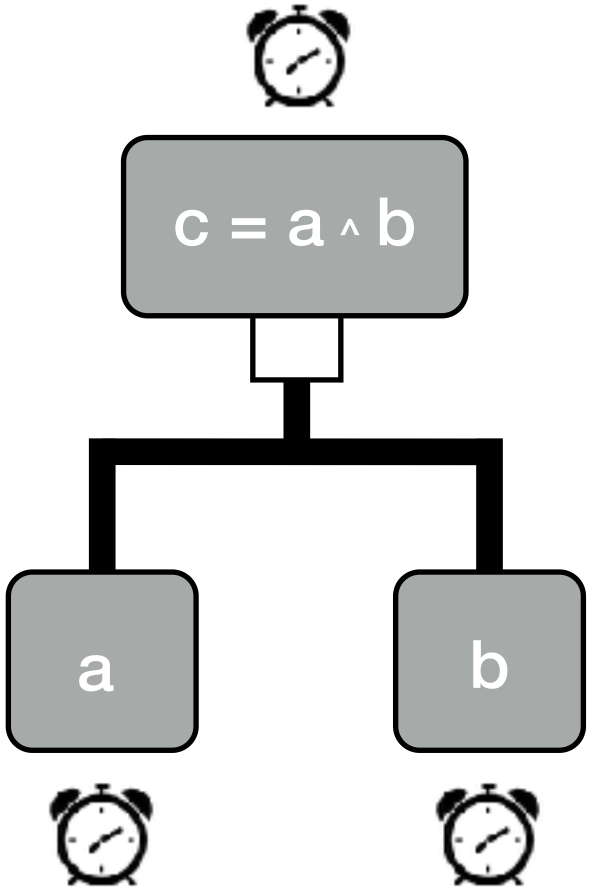
- **Overwriting:** loss of values

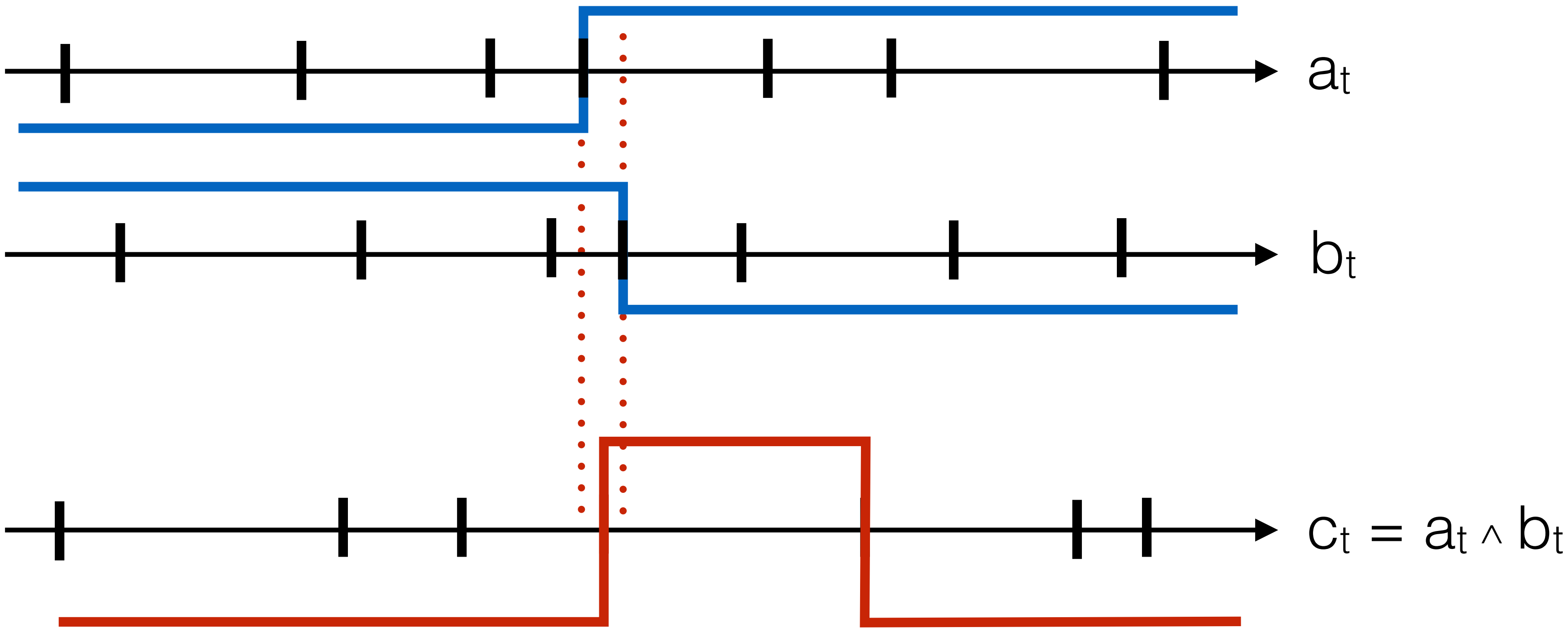


- **Oversampling:** duplication of values



- **Combination of signals**





Example from [Cas00]

Programming-based Approach

A program is a formal model ...

- precisely model every detail of the system
- based on the semantics of the programming language

Programming-based Approach

A program is a formal model ...

- precisely model every detail of the system
- based on the semantics of the programming language

... that can be executed and verified

- tests / simulations
- automated verification tools

Programming-based Approach

A program is a formal model ...

- precisely model every detail of the system
- based on the semantics of the programming language

... that can be executed and verified

- tests / simulations
- automated verification tools

Which programming language?

Synchronous Languages

Domain specific languages for reactive systems
[Benveniste, Berry, Caspi, Edwards, Halbwachs, Le Guernic, Pouzet ...]

A synchronous program executes in a **succession of discrete steps**
The programmer writes high-level specifications: **stream functions** à la Lustre

Based on **discrete logical time**, they offer:

- Mathematically precise semantics
- Efficient and reliable code generation
- Dedicated verification tools

Synchronous Languages

Domain specific languages for reactive systems
[Benveniste, Berry, Caspi, Edwards, Halbwachs, Le Guernic, Pouzet ...]

A synchronous program executes in a **succession of discrete steps**
The programmer writes high-level specifications: **stream functions** à la Lustre

Based on **discrete logical time**, they offer:

- Mathematically precise semantics
- Efficient and reliable code generation
- Dedicated verification tools

Scade/Lustre is routinely used in the industry

Synchronous Languages

Domain specific languages for reactive systems
[Benveniste, Berry, Caspi, Edwards, Halbwachs, Le Guernic, Pouzet ...]

A synchronous program executes in a **succession of discrete steps**
The programmer writes high-level specifications: **stream functions** à la Lustre

Based on **discrete logical time**, they offer:

- Mathematically precise semantics
- Efficient and reliable code generation
- Dedicated verification tools

Scade/Lustre is routinely used in the industry

Ideal framework to study quasi-periodic systems

Synchronous Languages

Domain specific languages for reactive systems
[Benveniste, Berry, Caspi, Edwards, Halbwachs, Le Guernic, Pouzet ...]

A synchronous program executes in a **succession of discrete steps**
The programmer writes high-level specifications: **stream functions** à la Lustre

Based on **discrete logical time**, they offer:

- Mathematically precise semantics
- Efficient and reliable code generation
- Dedicated verification tools

Scade/Lustre is routinely used in the industry

Ideal framework to study quasi-periodic systems

However for quasi-periodic systems:

- **Multiple synchronous programs** execute in parallel
- They are **not synchronized**
- The architecture is characterized by **real-time parameters**

Zélus: Lustre + ODEs

A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

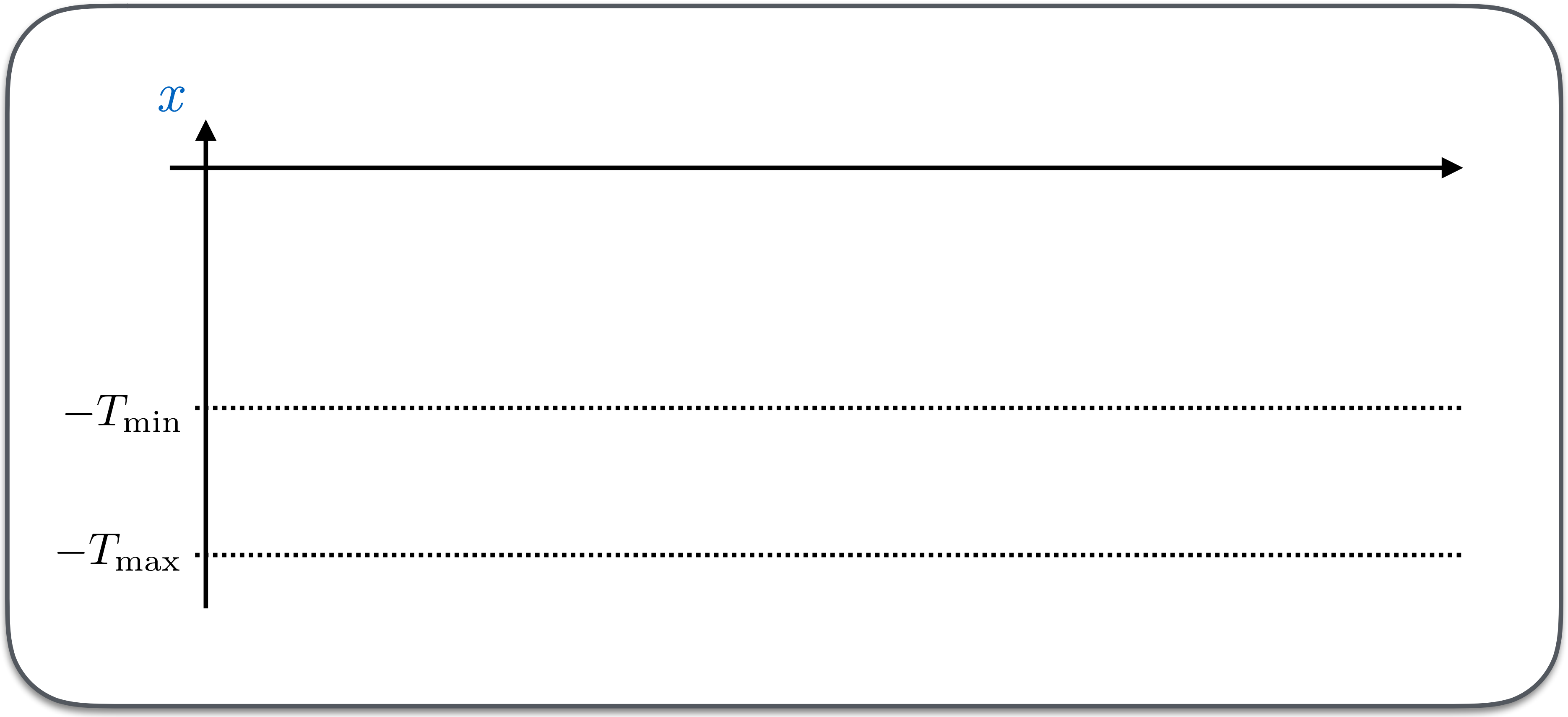
Zélus: Lustre + ODEs

A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```



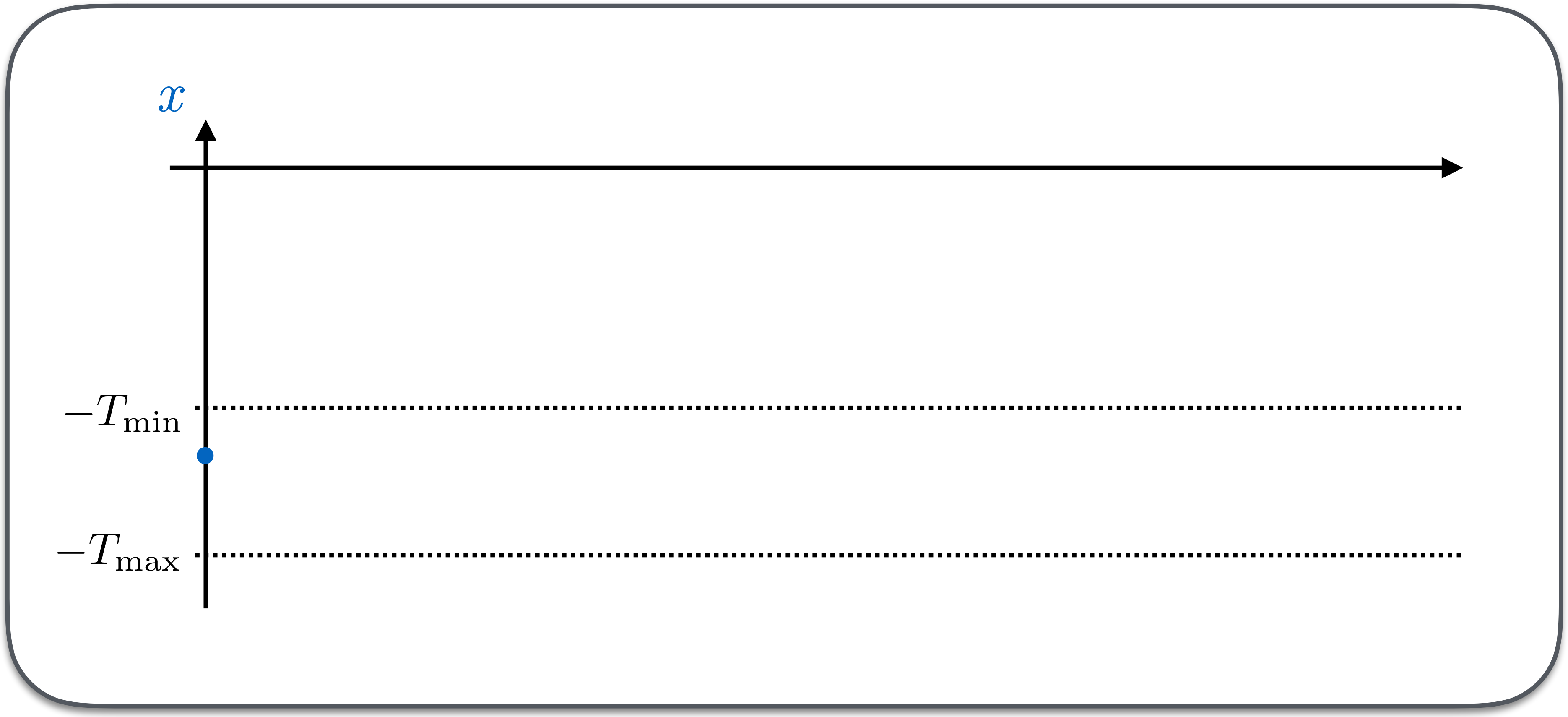
Zélus: Lustre + ODEs

A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```



http://zelus.di.ens.fr

Zélus: Lustre + ODEs

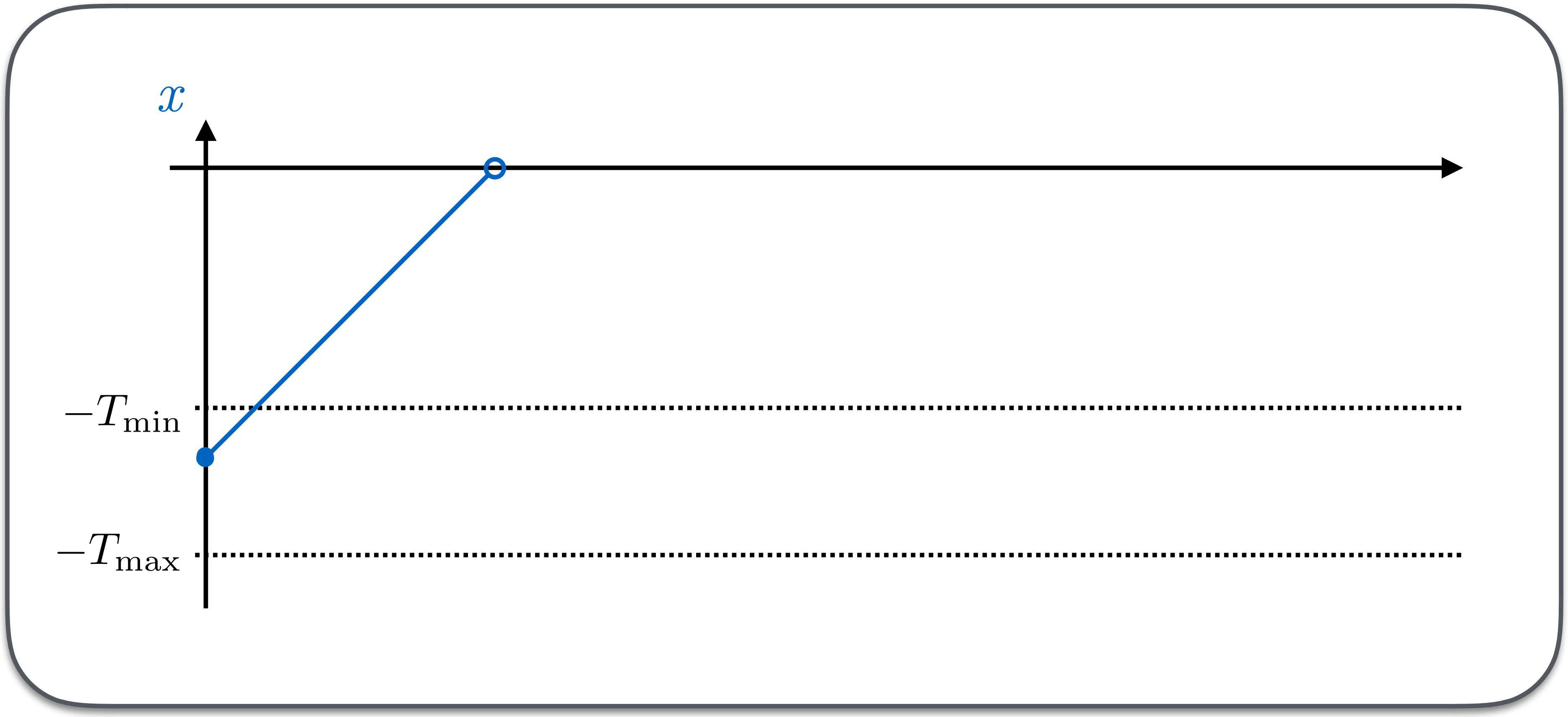
A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

dx/dt = 1 : time elapsing



http://zelus.di.ens.fr

Zélus: Lustre + ODEs

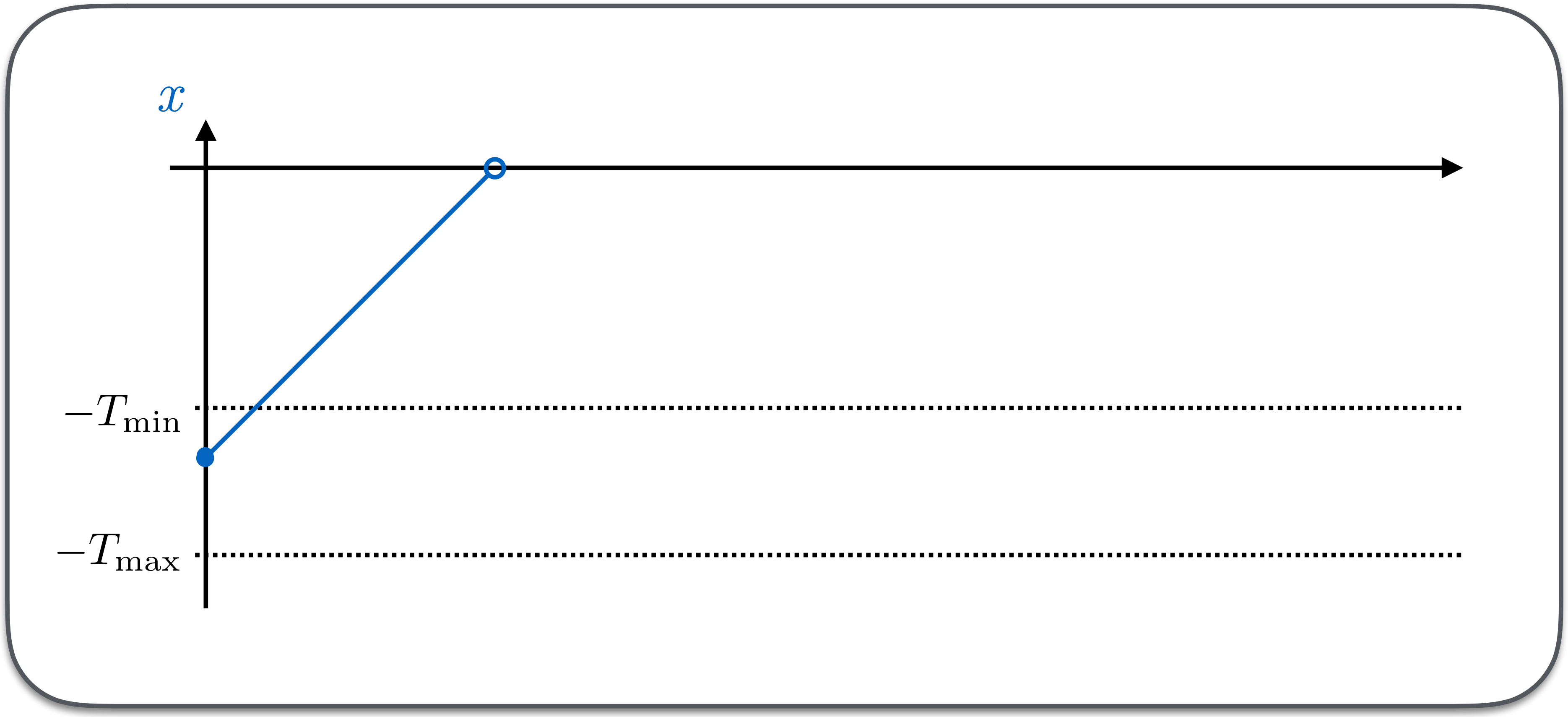
A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

dx/dt = 1 : time elapsing



Zélus: Lustre + ODEs

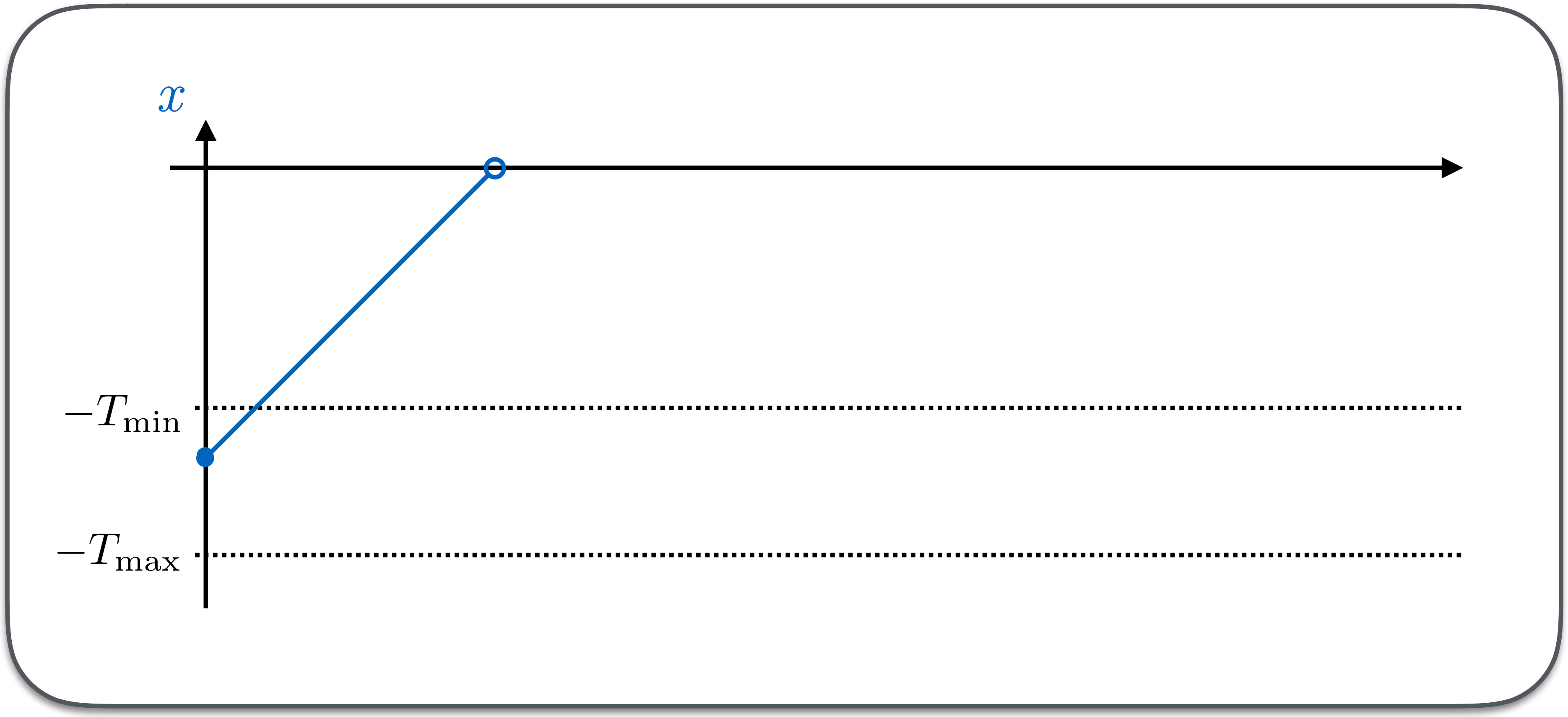
A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

dx/dt = 1 : time elapsing



Zélus: Lustre + ODEs

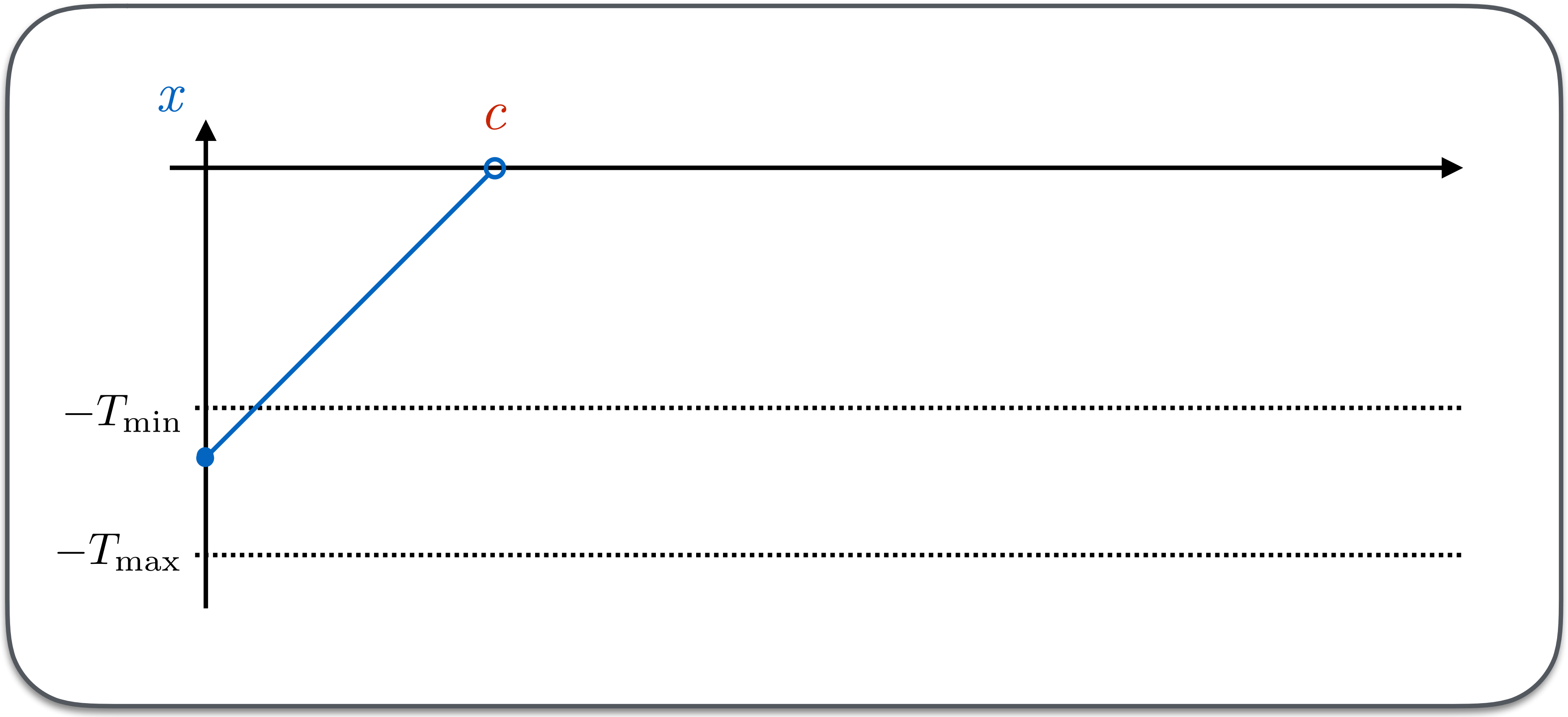
A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

dx/dt = 1 : time elapsing



Zélus: Lustre + ODEs

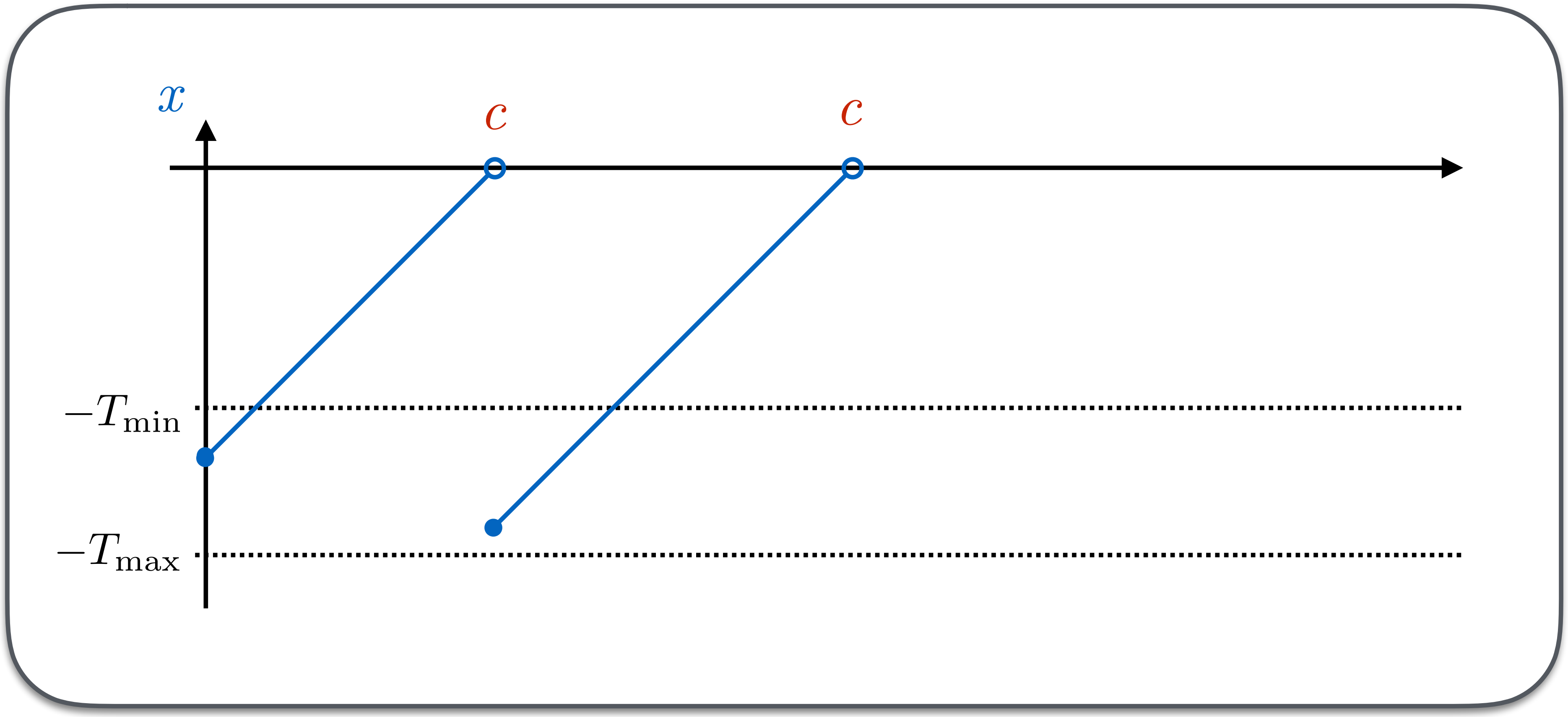
A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
        reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

dx/dt = 1 : time elapsing



Zélus: Lustre + ODEs

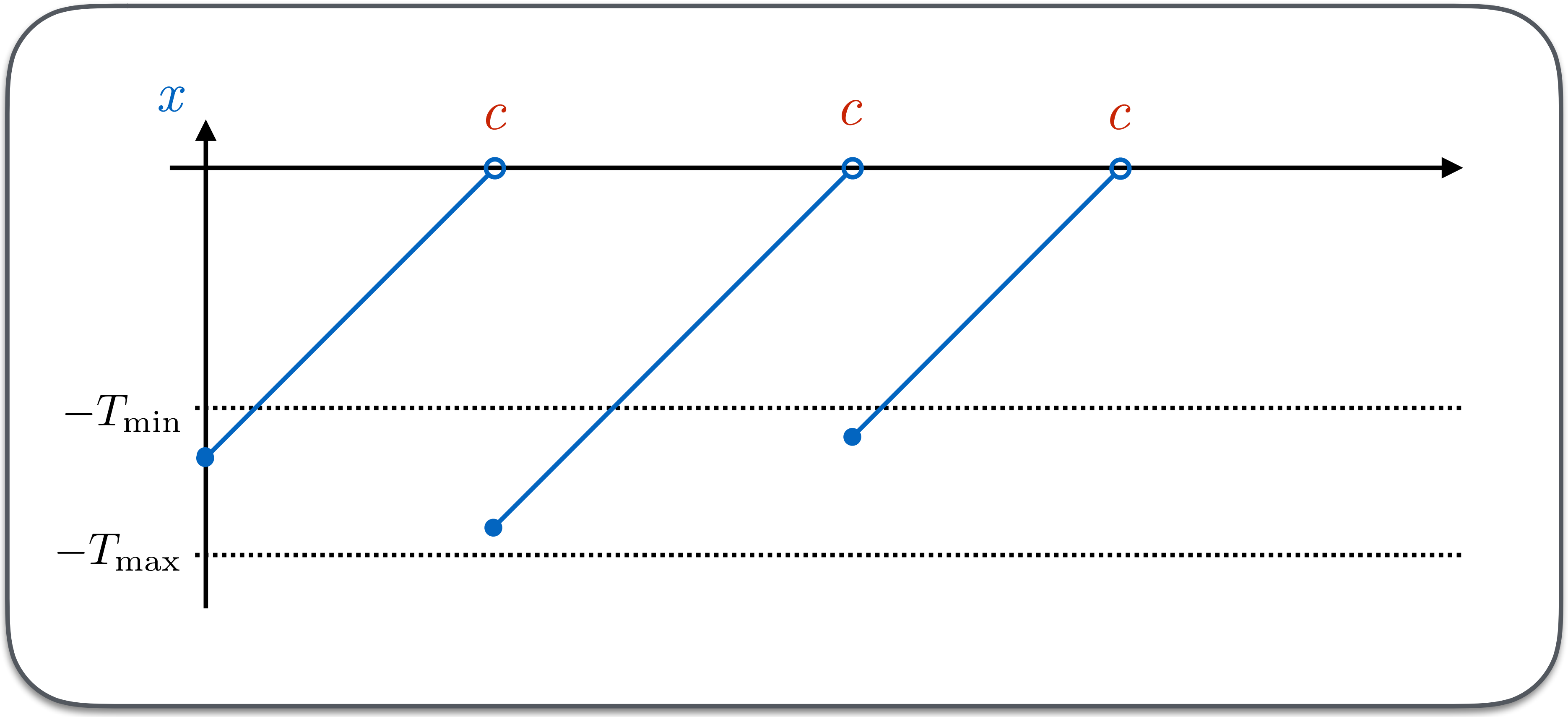
A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

dx/dt = 1 : time elapsing



Zélus: Lustre + ODEs

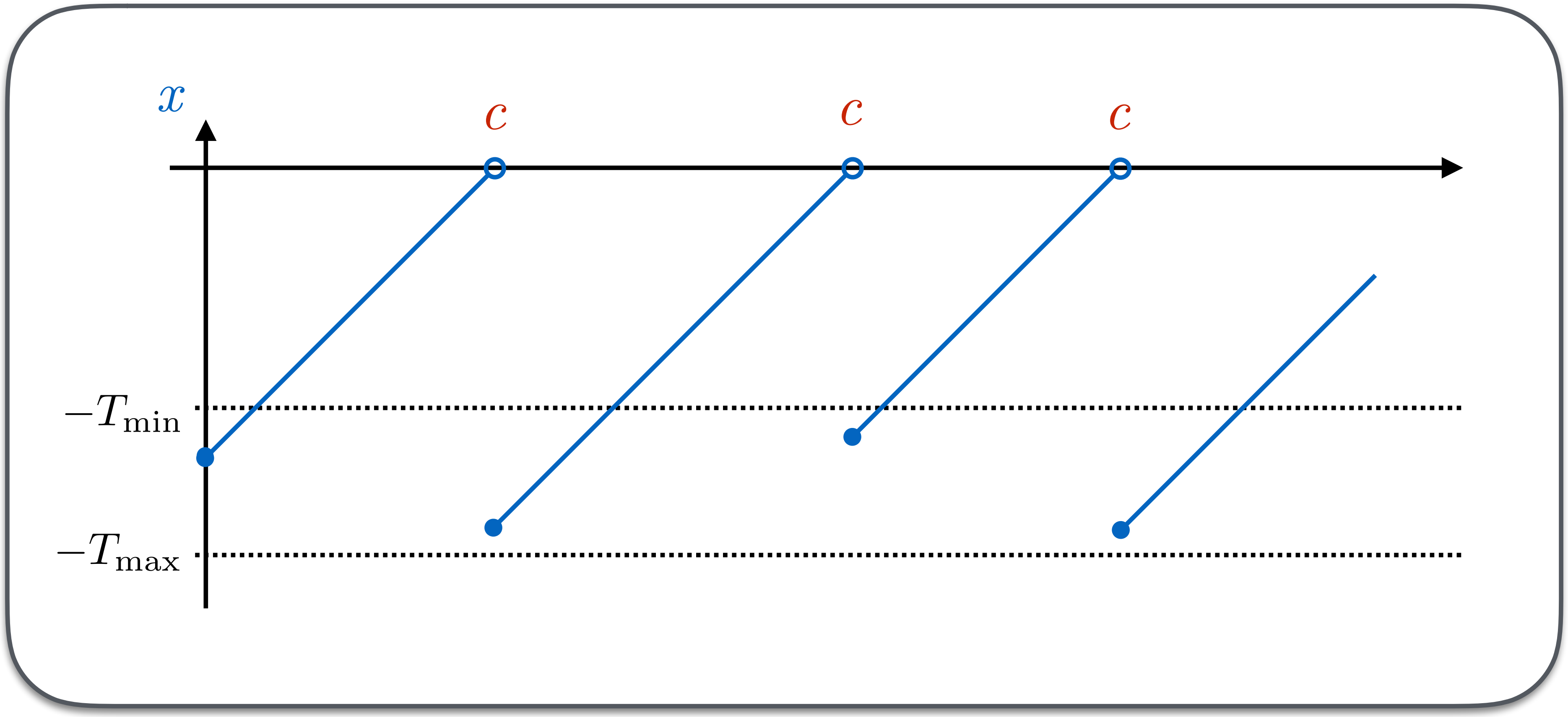
A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

dx/dt = 1 : time elapsing



Zélus: Lustre + ODEs

A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

$dx/dt = 1$: time elapsing

Zélus: Lustre + ODEs

A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

$dx/dt = 1$: time elapsing

Discrete controllers are activated on signal emissions

```
let hybrid rt_controller(sensor1, sensor2) = o where
  rec c1 = metro(t_min, t_max)
  and c2 = metro(t_min, t_max)
  and present c1 → do emit cmd1 = fgs(sensor1) done
  and present c2 → do emit cmd2 = fgs(sensor2) done
  and cmd = if switch then cmd1 else cmd2
```

Zélus: Lustre + ODEs

A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

dx/dt = 1 : time elapsing

Discrete controllers are activated on signal emissions

```
let hybrid rt_controller(sensor1, sensor2) = o where
  rec c1 = metro(t_min, t_max)
  and c2 = metro(t_min, t_max)
  and present c1 → do emit cmd1 = fgs(sensor1) done
  and present c2 → do emit cmd2 = fgs(sensor2) done
  and cmd = if switch then cmd1 else cmd2
```

Design discrete controllers and the real-time architecture
in the very same language.

Zélus: Lustre + ODEs

A synchronous language extended with continuous time

[Benveniste, Bourke, Caillaud, Pouzet]

Continuous-time dynamics of the architecture simulated with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

dx/dt = 1 : time elapsing

Discrete controllers are activated on signal emissions

```
let hybrid rt_controller(sensor1, sensor2) = o where
  rec c1 = metro(t_min, t_max)
  and c2 = metro(t_min, t_max)
  and present c1 → do emit cmd1 = fgs(sensor1) done
  and present c2 → do emit cmd2 = fgs(sensor2) done
  and cmd = if switch then cmd1 else cmd2
```

Same approach in
Ptolemy [Lee]
Simulink [Mathworks]

Design discrete controllers and the real-time architecture
in the very same language.

Contributions

Verification

Verifying safety properties of quasi-periodic systems

The Quasi-Synchronous Abstraction

Implementation

Deploying code on quasi-periodic architectures

Loosely Time-Triggered Architectures

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

Contributions

Verification

Verifying safety properties of quasi-periodic systems

The Quasi-Synchronous Abstraction

Implementation

Deploying code on quasi-periodic architectures

Loosely Time-Triggered Architectures

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

Abstraction is not sound in general

Give exact conditions of application

Generalization to multirate systems

Contributions

Verification

Verifying safety properties of quasi-periodic systems

The Quasi-Synchronous Abstraction

Abstraction is not sound in general

Give exact conditions of application

Generalization to multirate systems

Implementation

Deploying code on quasi-periodic architectures

Loosely Time-Triggered Architectures

Unified synchronous framework

Executable specifications

Correctness proofs

Optimizations and comparisons

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

Contributions

Verification

Verifying safety properties of quasi-periodic systems

The Quasi-Synchronous Abstraction

Abstraction is not sound in general

Give exact conditions of application

Generalization to multirate systems

Implementation

Deploying code on quasi-periodic architectures

Loosely Time-Triggered Architectures

Unified synchronous framework

Executable specifications

Correctness proofs

Optimizations and comparisons

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

Zélus extended with timed nondeterminism

Symbolic simulation

Modular source-to-source compilation

Prototype implementation

Overview

Verification

Verifying safety properties of quasi-periodic systems

Quasi-Synchronous Abstraction

Contributions

Abstraction is not sound in general

Give exact conditions of application

Generalization to multirate systems

Overview

Verification

Verifying safety properties of quasi-periodic systems

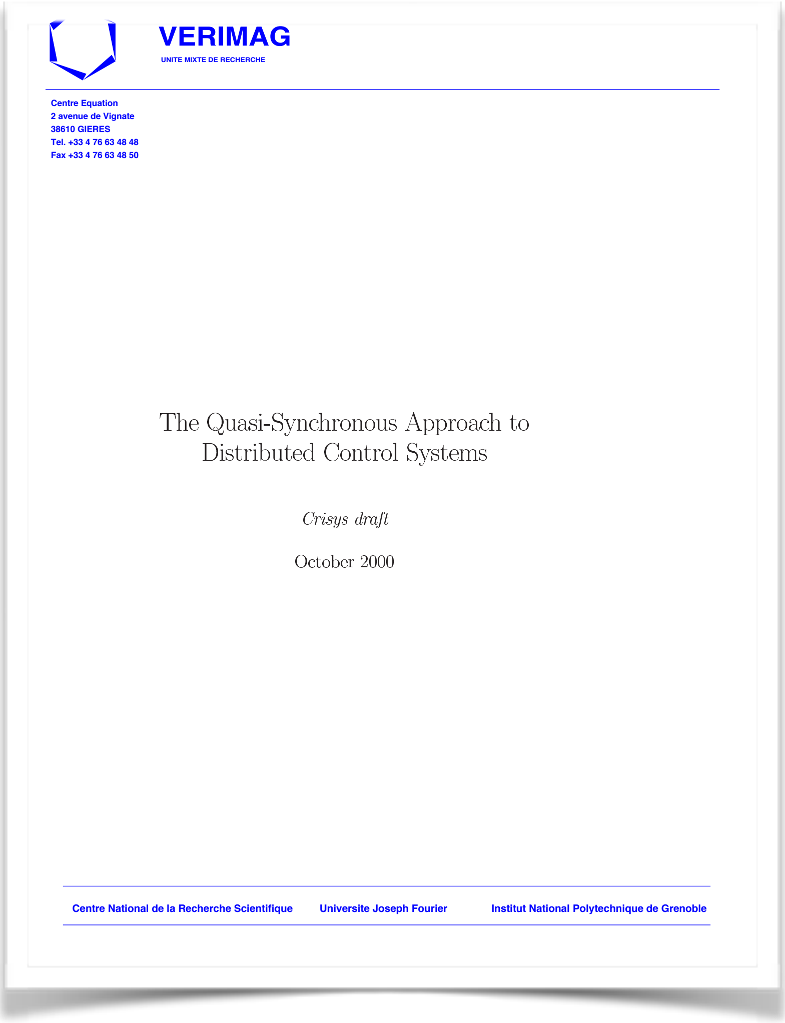
Quasi-Synchronous Abstraction

Contributions

Abstraction is not sound in general

Give exact conditions of application

Generalization to multirate systems



VERIMAG

UNITE MIXTE DE RECHERCHE

Centre Equation
2 avenue de Vignate
38610 GIERES
Tel. +33 4 76 63 48 48
Fax +33 4 76 63 48 50

The Quasi-Synchronous Approach to
Distributed Control Systems

Crisys draft

October 2000

Centre National de la Recherche Scientifique   Universite Joseph Fourier   Institut National Polytechnique de Grenoble

Industrial practices observed at Airbus

[Cas00]

Overview

Verification

Verifying safety properties of quasi-periodic systems

Quasi-Synchronous Abstraction

Contributions

Abstraction is not sound in general

Give exact conditions of application

Generalization to multirate systems

Industrial practices observed at Airbus

[Bhattacharyya, Halbwachs, Jahier, Mandel, Miller, Tinelli, Larrieu, Raymond, Shankar, ...]

[HM06, JHR08, BMY+14, LS14, LGS15, BMY+15]

[Cas00]

Overview

Verification

Verifying safety properties of quasi-periodic systems

Quasi-Synchronous Abstraction

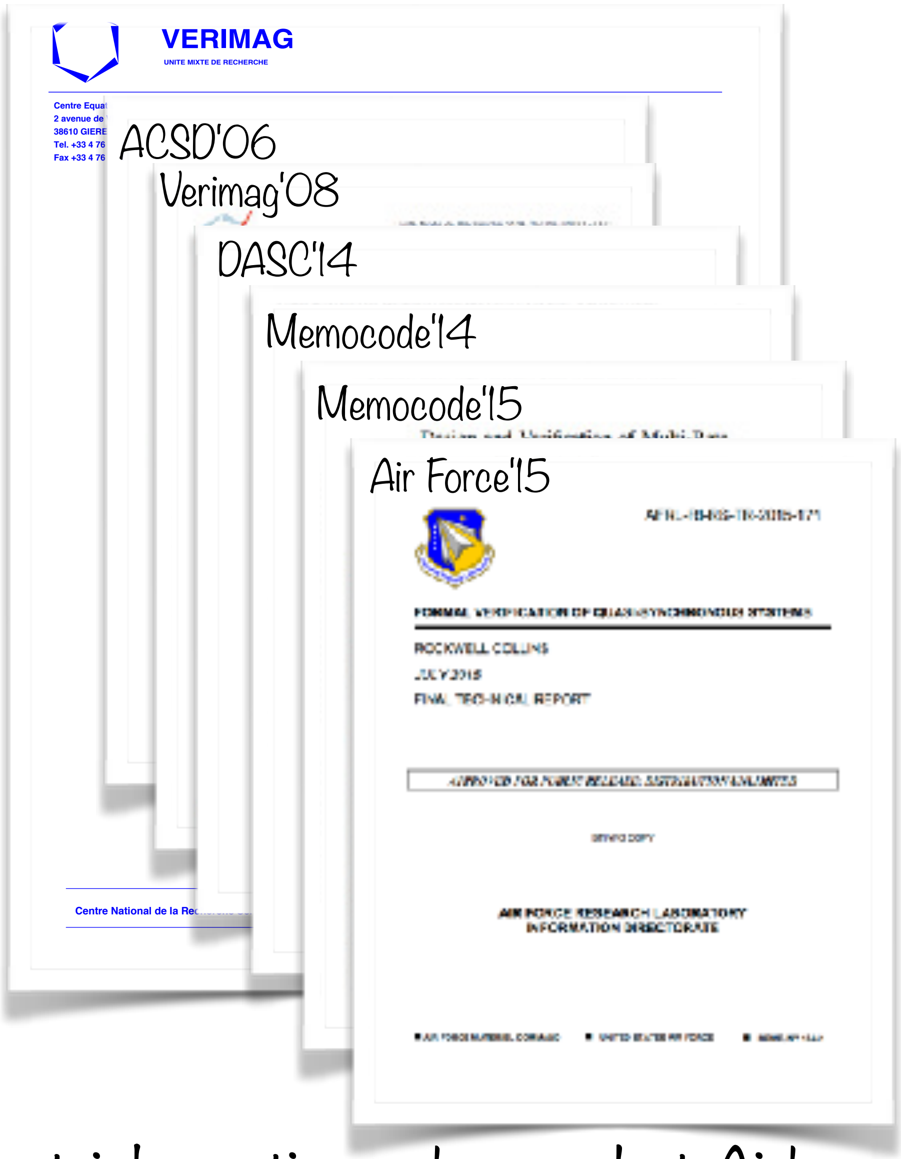
Contributions

Abstraction is not sound in general

Give exact conditions of application

Generalization to multirate systems

Is the abstraction sound?



Industrial practices observed at Airbus

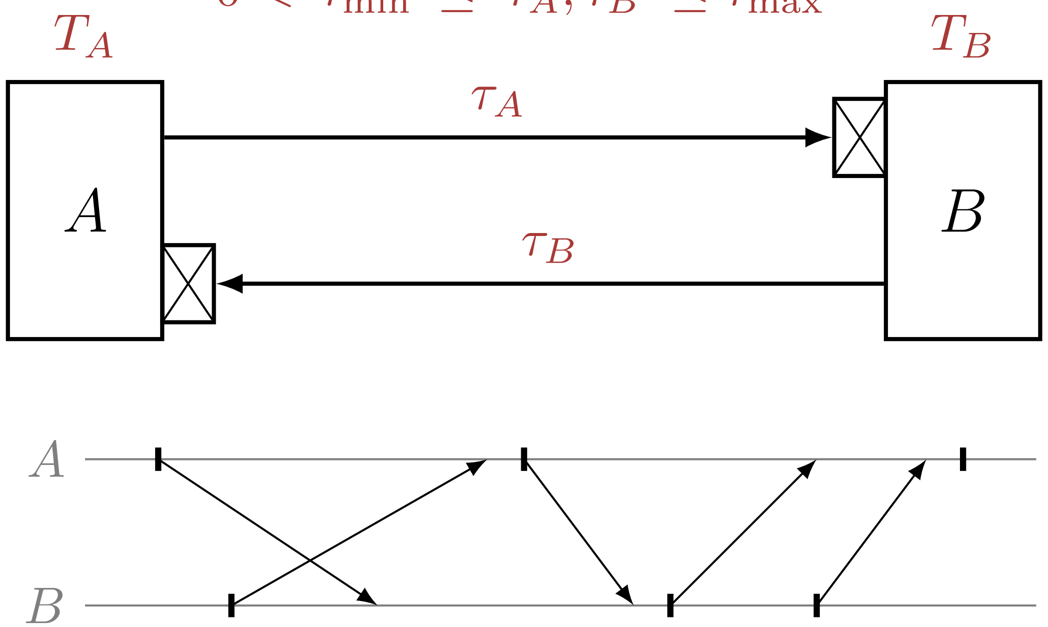
[Bhattacharyya, Halbwachs, Jahier, Mandel, Miller, Tinelli, Larrieu, Raymond, Shankar, ...]

[HM06, JHR08, BMY+14, LS14, LGS15, BMY+15]

[Cas00]

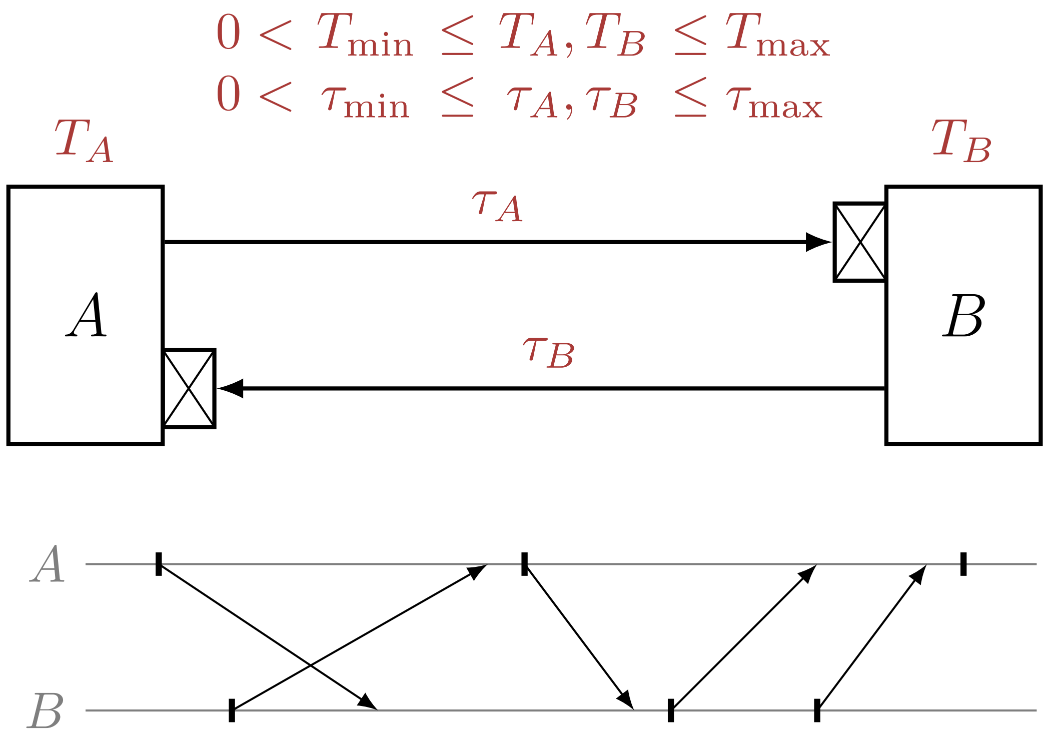
The Big Picture

$$0 < T_{\min} \leq T_A, T_B \leq T_{\max}$$
$$0 < \tau_{\min} \leq \tau_A, \tau_B \leq \tau_{\max}$$



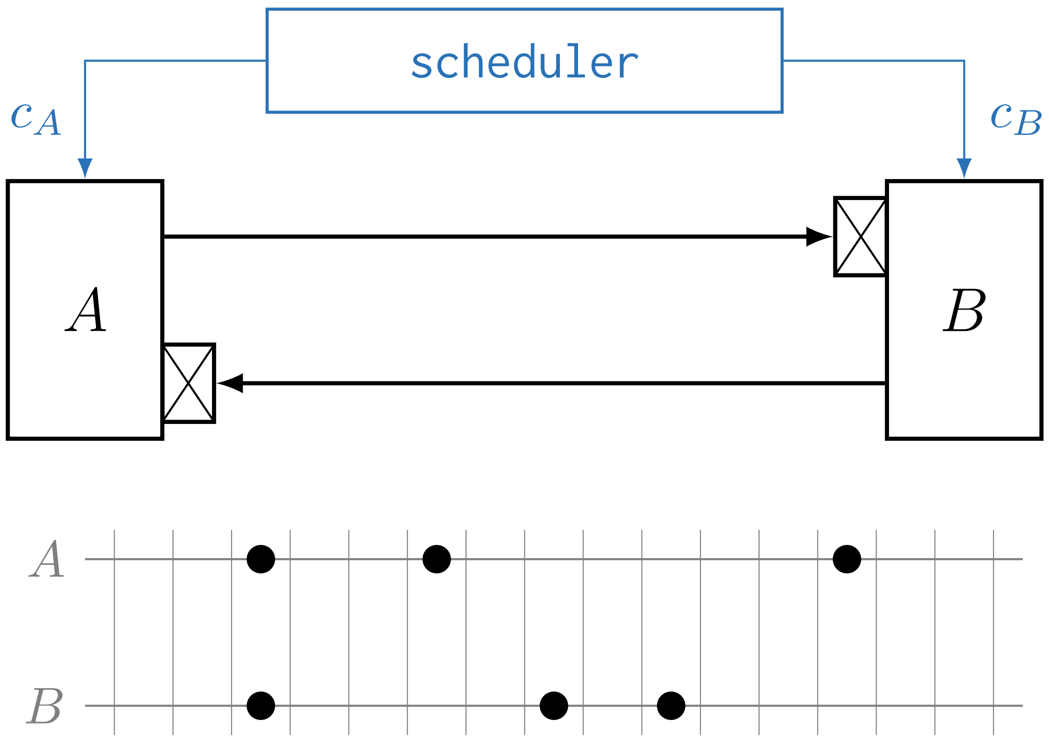
Real-time Model (RT)

The Big Picture



$$0 < T_{\min} \leq T_A, T_B \leq T_{\max}$$
$$0 < \tau_{\min} \leq \tau_A, \tau_B \leq \tau_{\max}$$

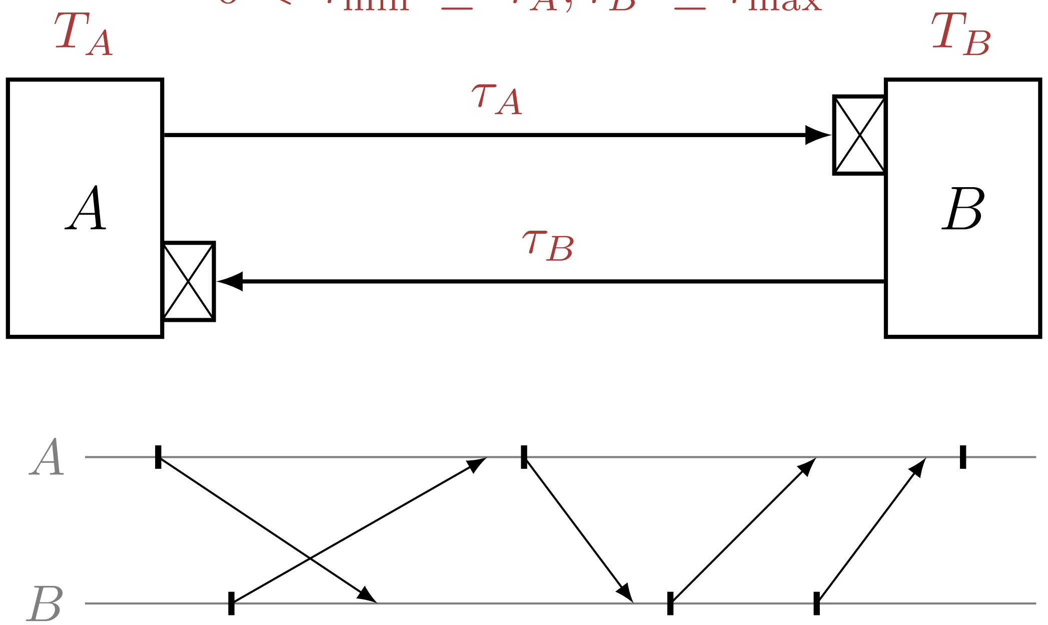
Real-time Model (RT)



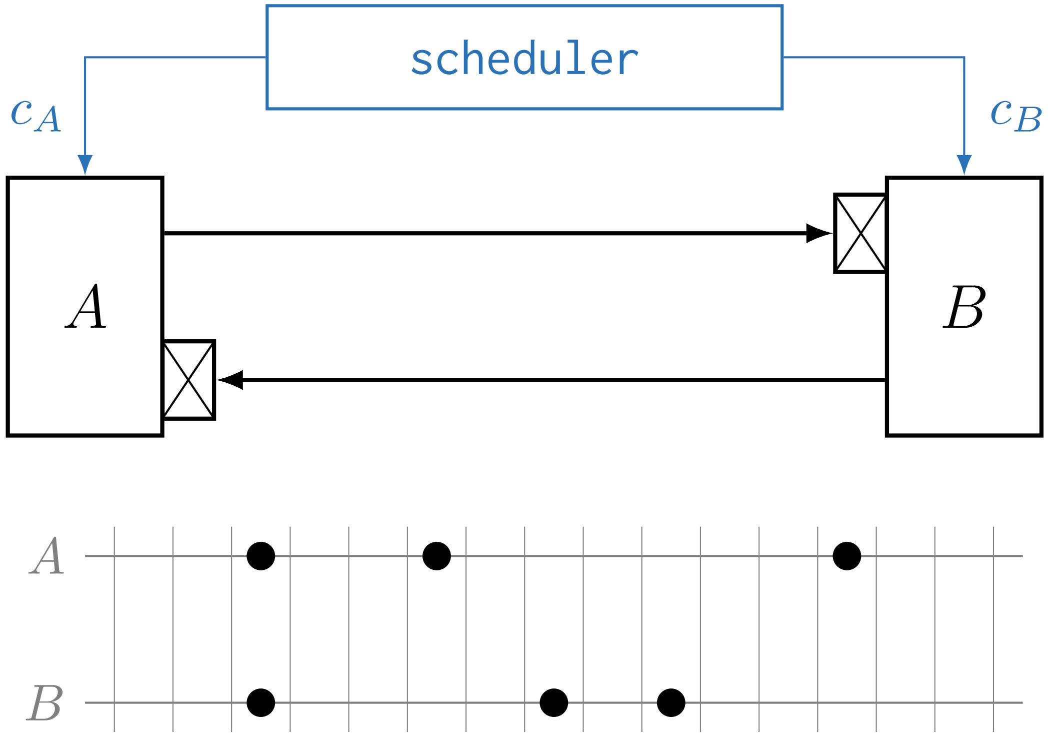
Discrete-time Model (DT)

The Big Picture

$$0 < T_{\min} \leq T_A, T_B \leq T_{\max}$$
$$0 < \tau_{\min} \leq \tau_A, \tau_B \leq \tau_{\max}$$



Real-time Model (RT)

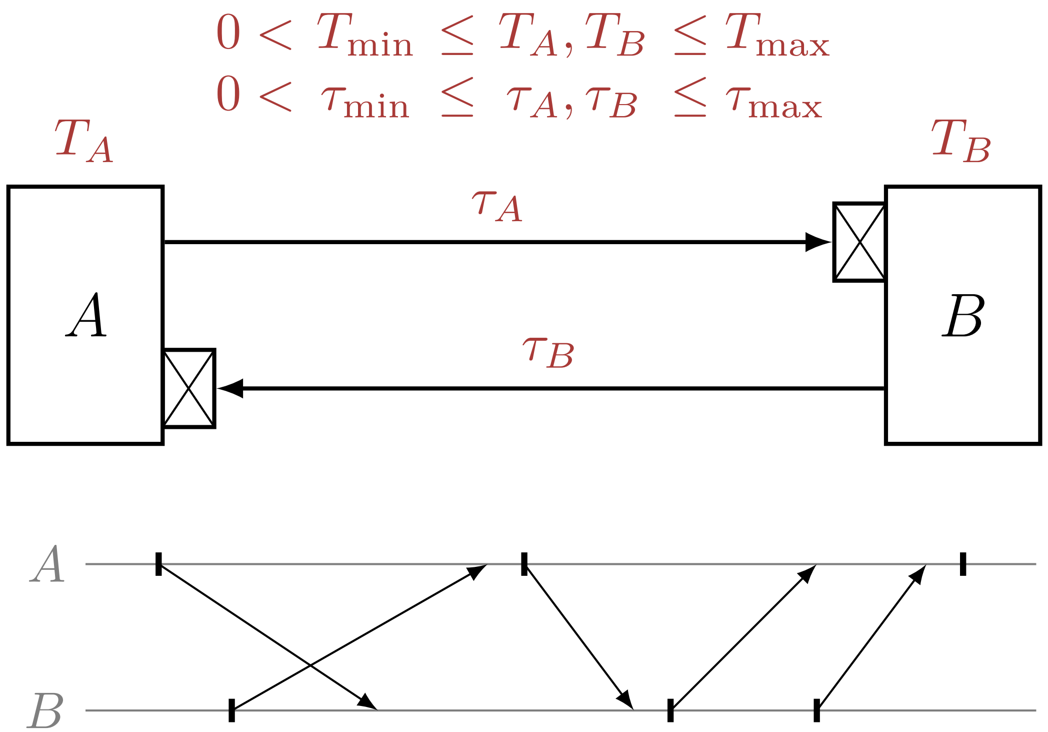


Discrete-time Model (DT)

$$RT \models \varphi \quad \Longleftarrow \quad DT \models \varphi$$

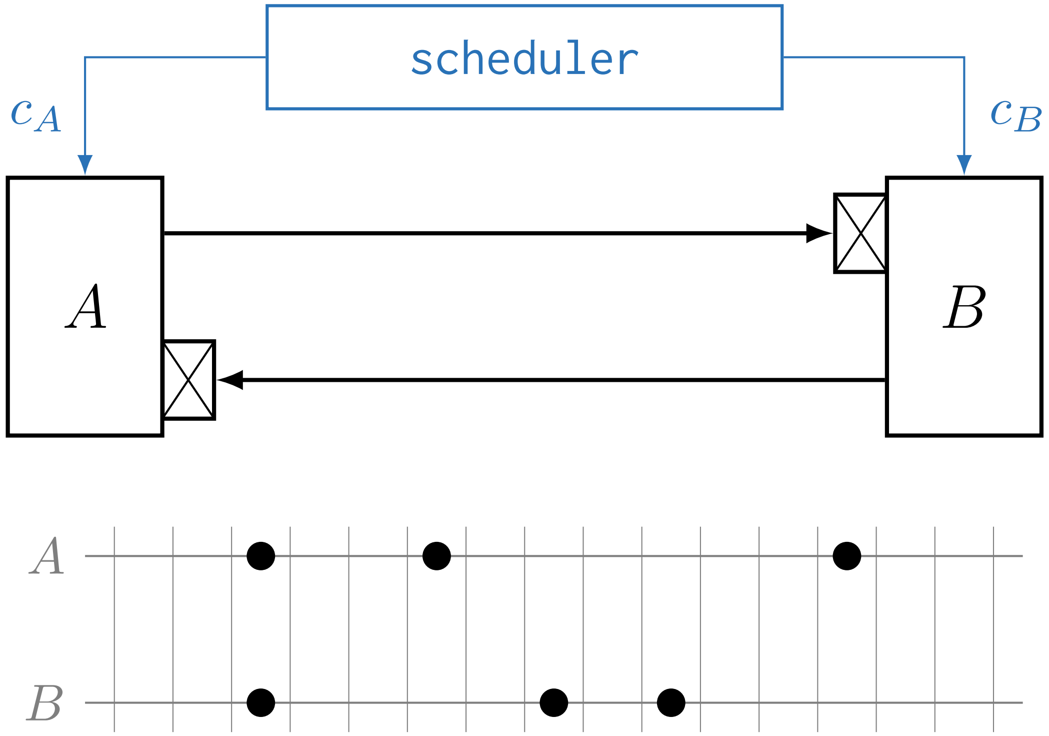
Soundness

The Big Picture



$$0 < T_{\min} \leq T_A, T_B \leq T_{\max}$$
$$0 < \tau_{\min} \leq \tau_A, \tau_B \leq \tau_{\max}$$

Real-time Model (RT)



Discrete-time Model (DT)

$$RT \models \varphi \xLongleftarrow[\text{Soundness}]{} DT \models \varphi$$

Why discretize?

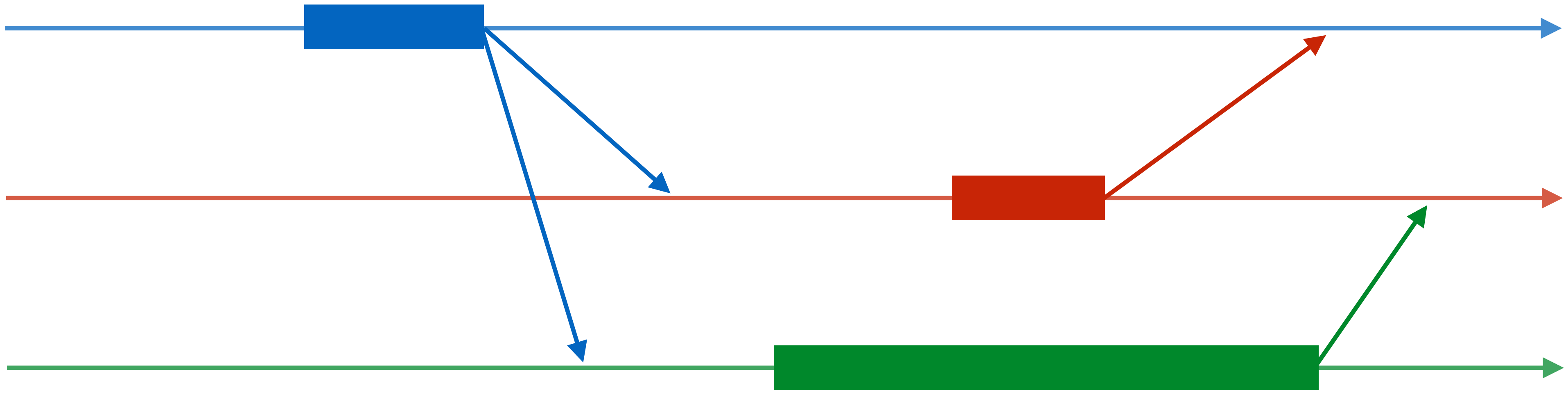
Verification in a simpler discrete-time model [Milner, Berry, Halbwachs, ...]

Use discrete-time model checking tools (Lesar-Verimag, Kind2-UIowa)

[Mil83, BS01, HB02, GG03a, GG03b, HM06]

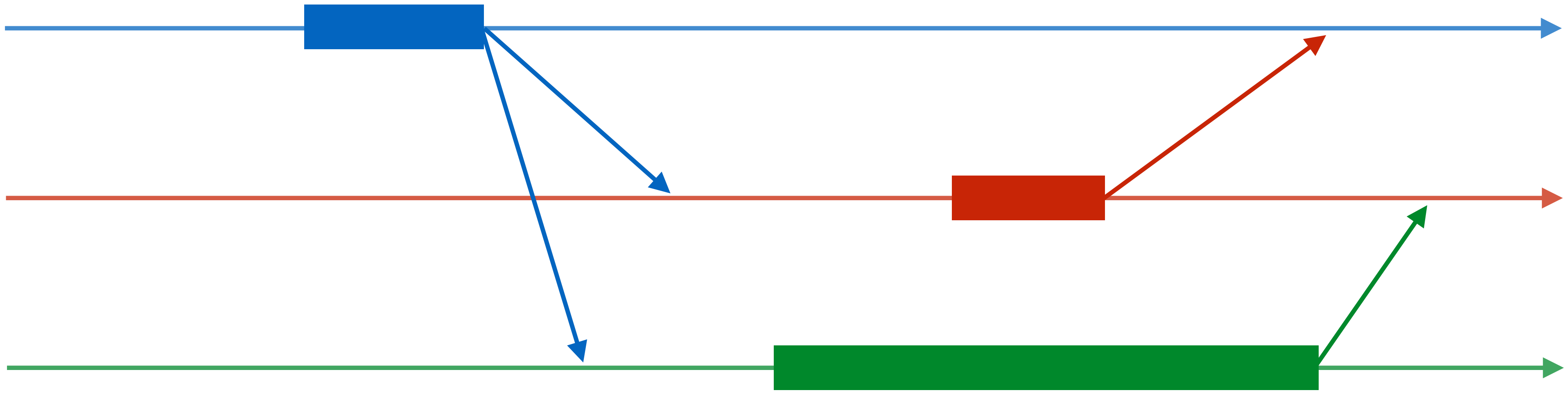
[HLR92, CMST16]

Abstracting Real Time



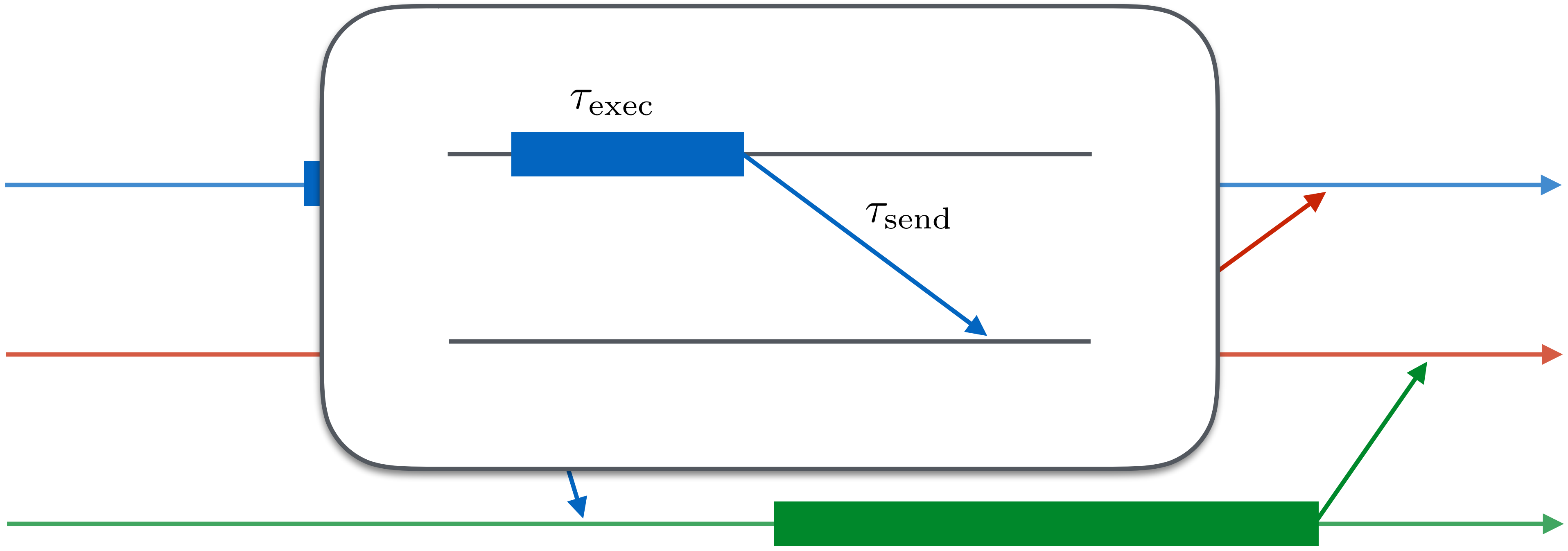
Abstracting Real Time

Abstracting execution time



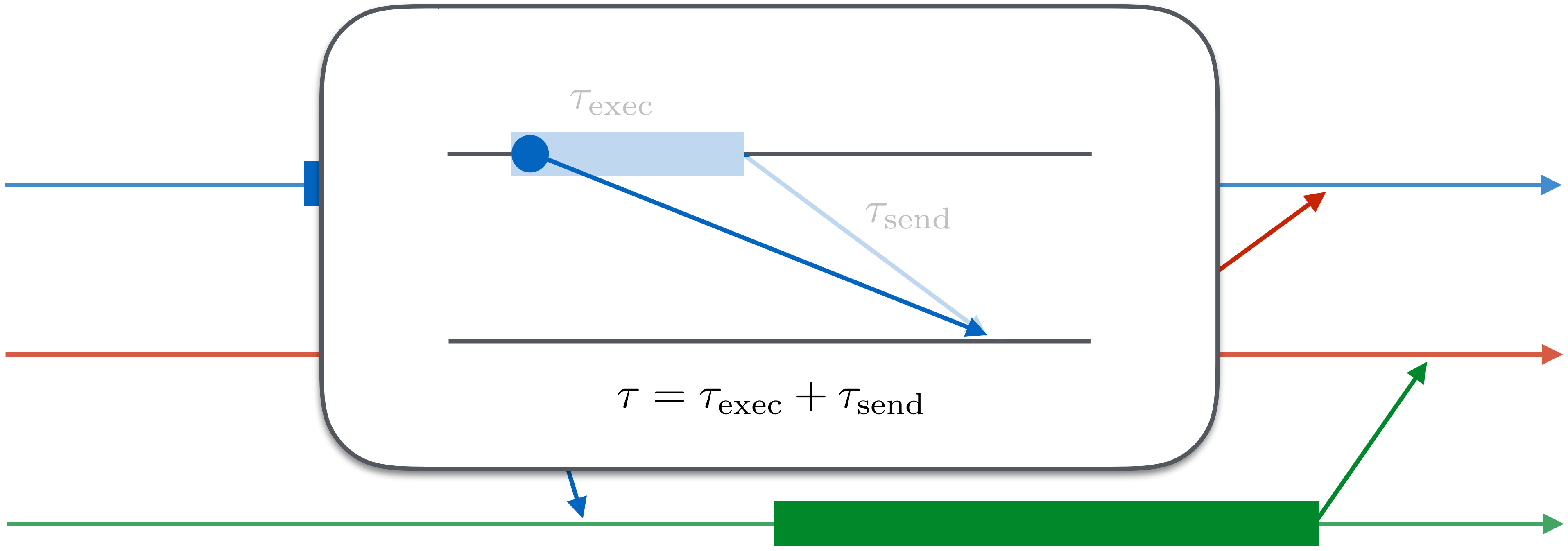
Abstracting Real Time

Abstracting execution time



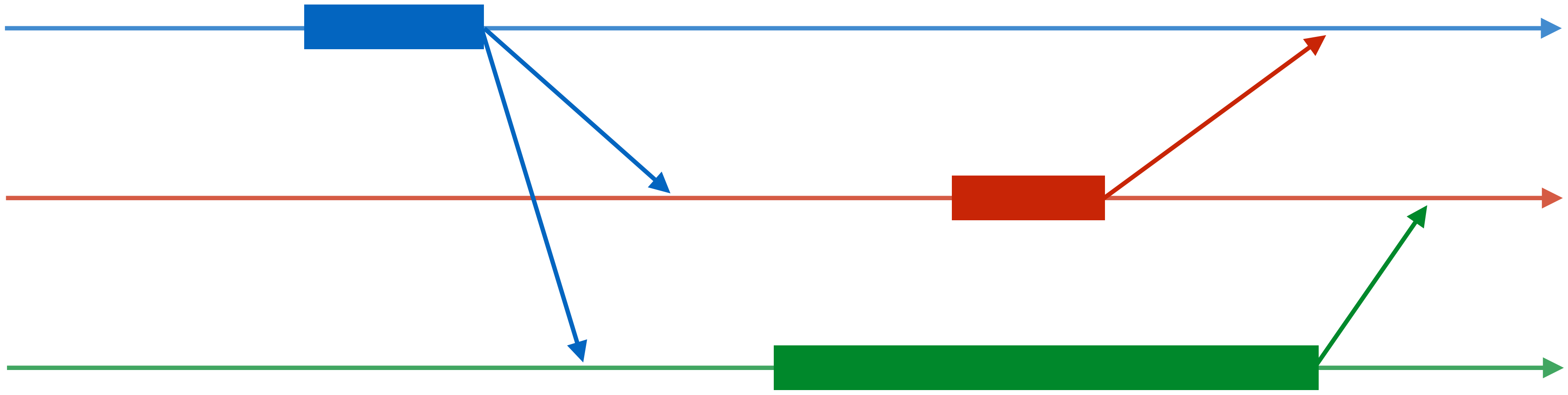
Abstracting Real Time

Abstracting execution time



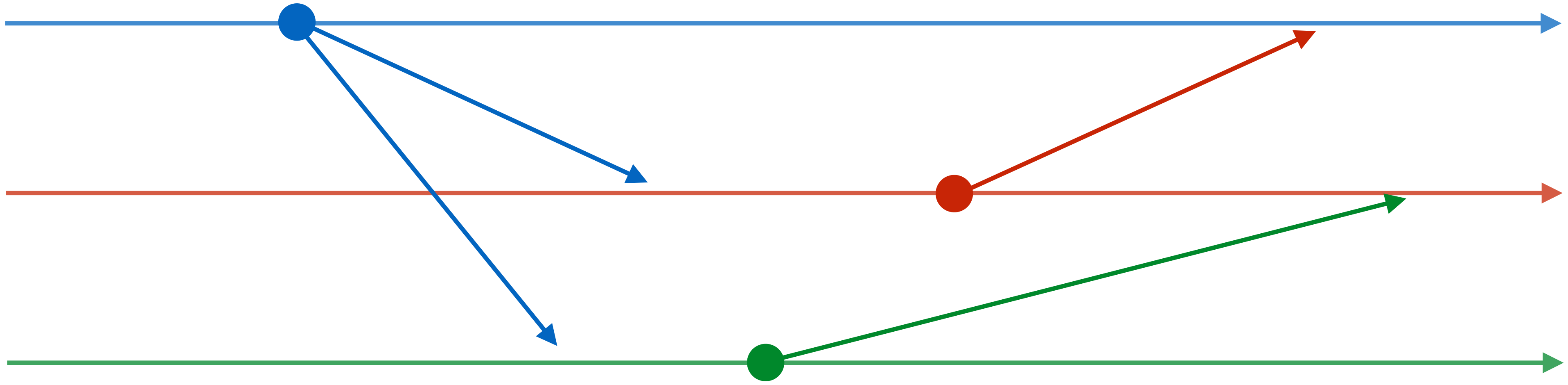
Abstracting Real Time

Abstracting execution time



Abstracting Real Time

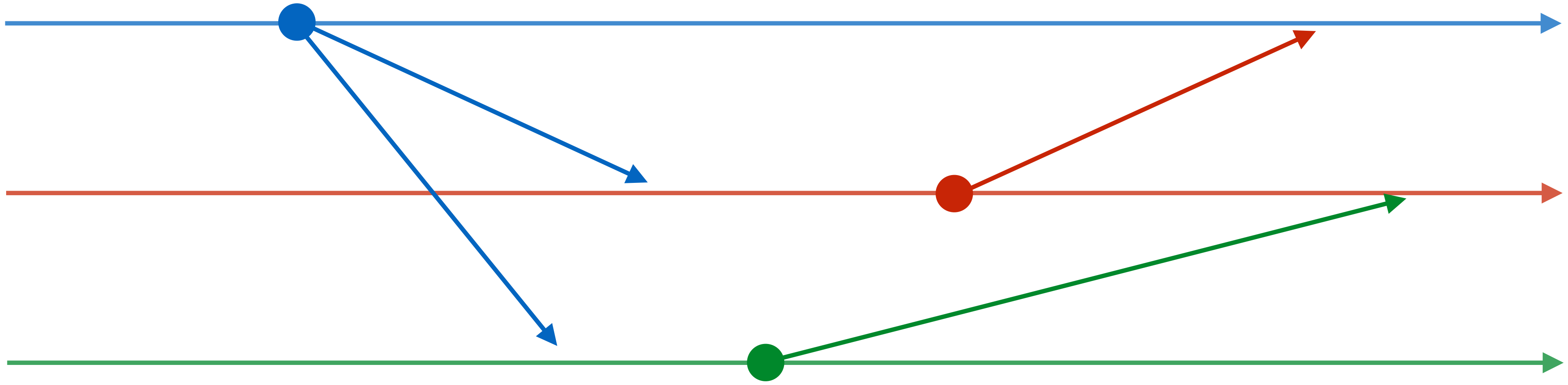
Abstracting execution time



Abstracting Real Time

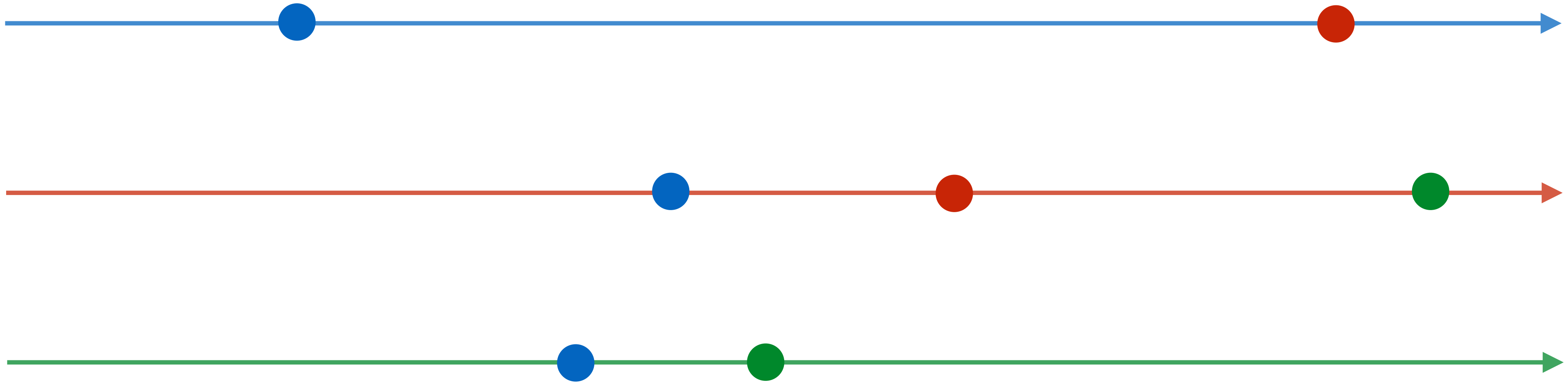
Abstracting execution time

Abstracting communication



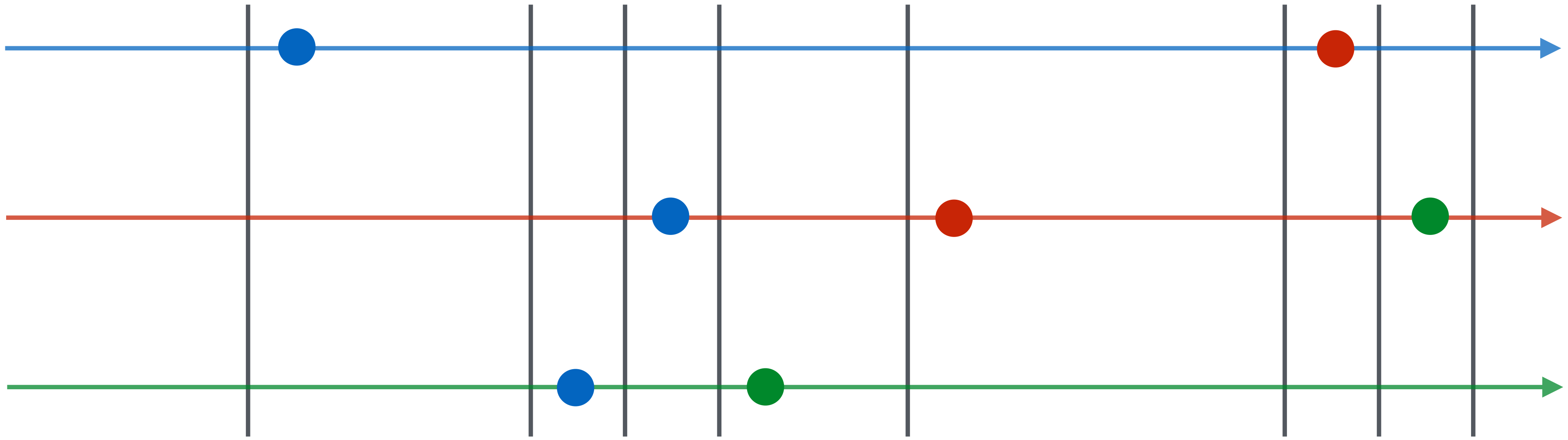
Abstracting Real Time

Abstracting execution time
Abstracting communication



Abstracting Real Time

Abstracting execution time
Abstracting communication



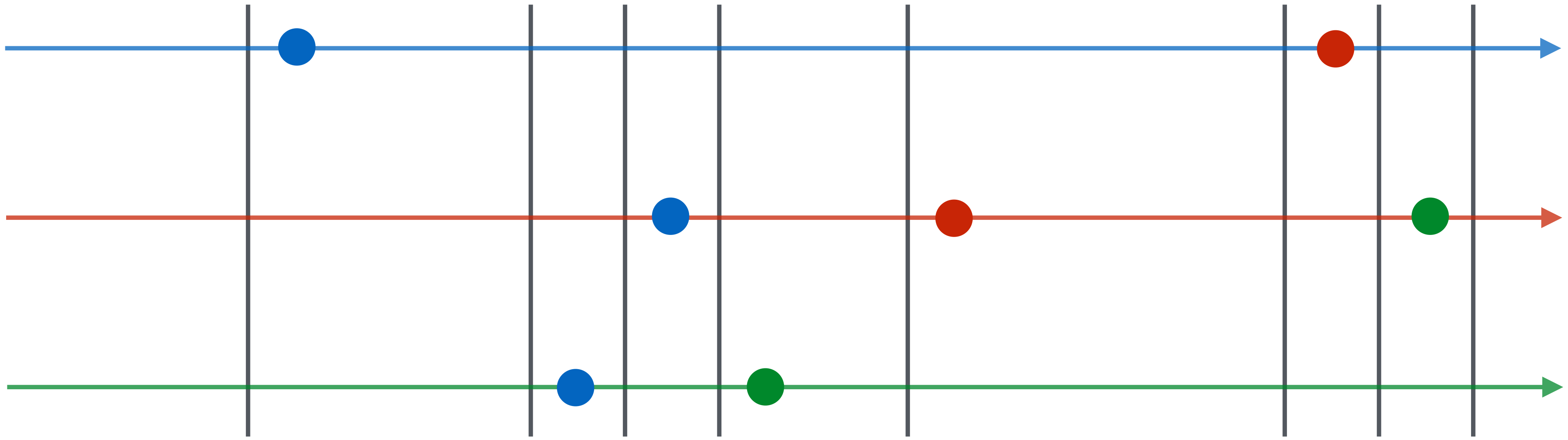
Abstracting Real Time

Abstracting execution time

Abstracting communication

Problems:

- Lots of possible interleavings
- Too general

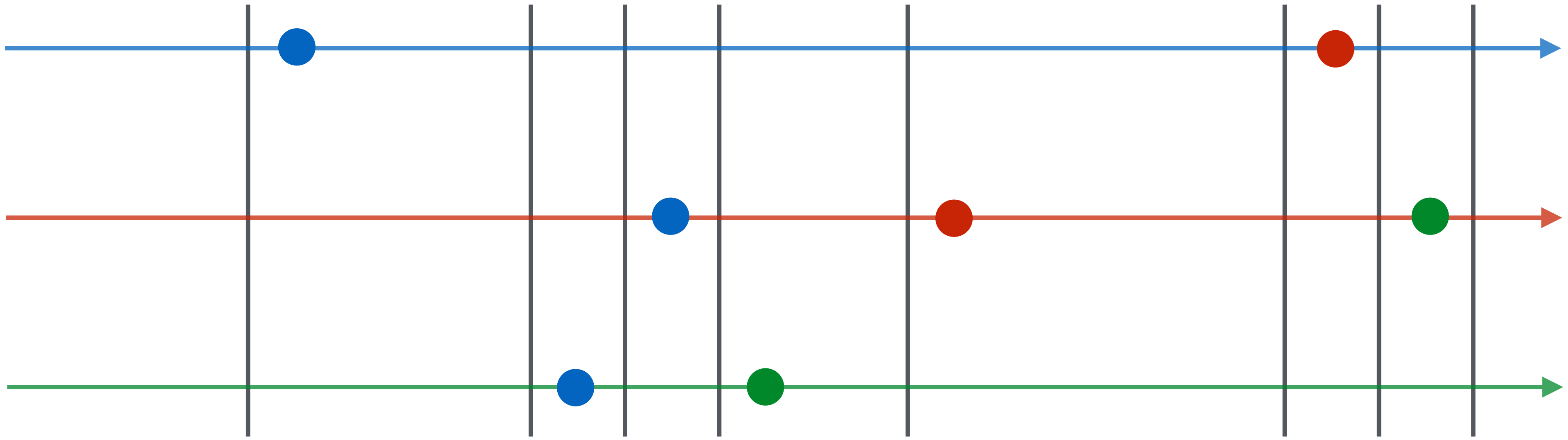


Abstracting Real Time

Abstracting execution time
Abstracting communication

Problems:

- Lots of possible interleavings
- Too general



Can we do better using real-time assumptions?

The Quasi-Synchronous Abstraction

Focus on **'almost' synchronous** architectures with **fast transmissions**

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

The Quasi-Synchronous Abstraction

Focus on **'almost' synchronous** architectures with **fast transmissions**

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

Reduce the state-space in two ways:

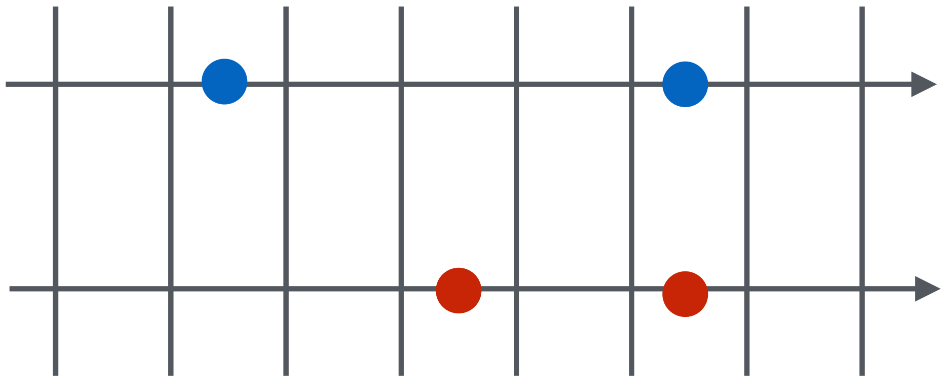
The Quasi-Synchronous Abstraction

Focus on **'almost' synchronous** architectures with **fast transmissions**

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

Reduce the state-space in two ways:

1. Transmissions as unit delays
 (one step of the logical clock)



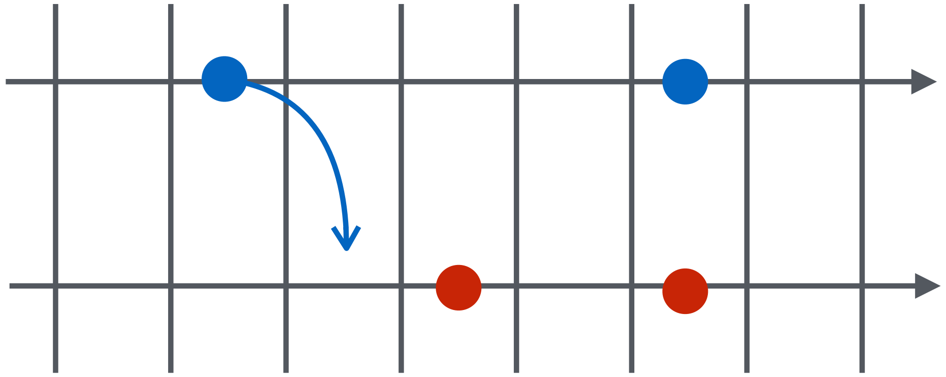
The Quasi-Synchronous Abstraction

Focus on **'almost' synchronous** architectures with **fast transmissions**

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

Reduce the state-space in two ways:

1. Transmissions as unit delays (one step of the logical clock)



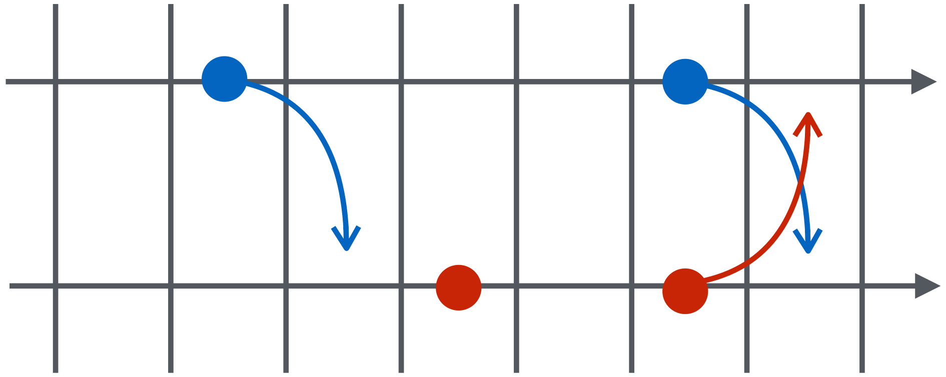
The Quasi-Synchronous Abstraction

Focus on **'almost' synchronous** architectures with **fast transmissions**

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

Reduce the state-space in two ways:

1. Transmissions as unit delays (one step of the logical clock)



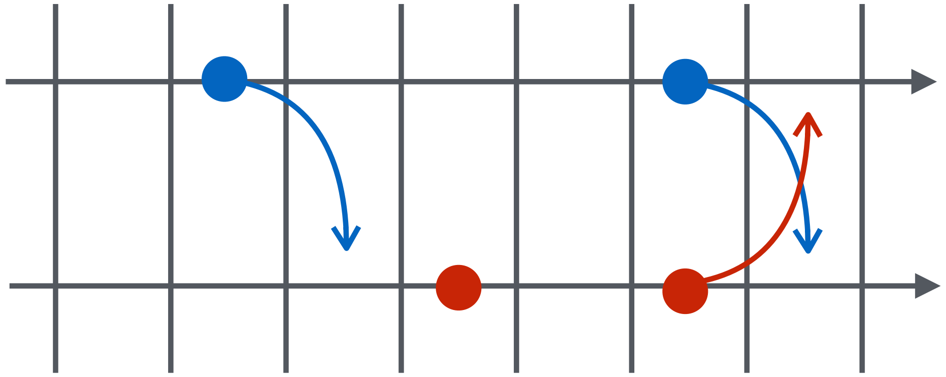
The Quasi-Synchronous Abstraction

Focus on **'almost' synchronous** architectures with **fast transmissions**

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

Reduce the state-space in two ways:

1. Transmissions as unit delays
 (one step of the logical clock)



Replace transmission with precedence

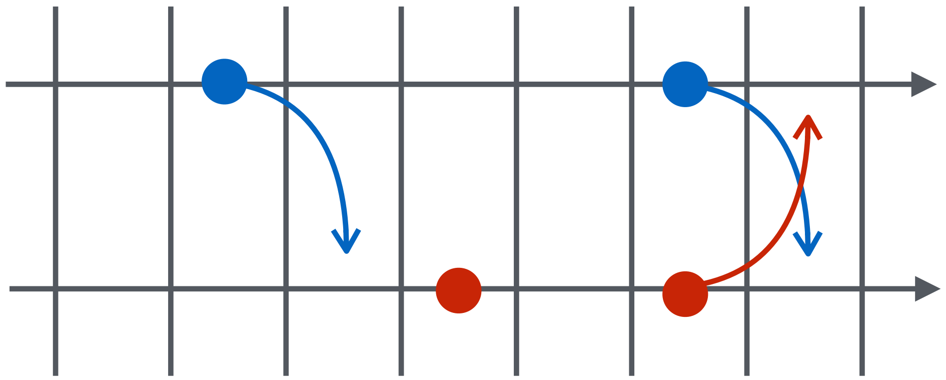
The Quasi-Synchronous Abstraction

Focus on **'almost' synchronous** architectures with **fast transmissions**

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

Reduce the state-space in two ways:

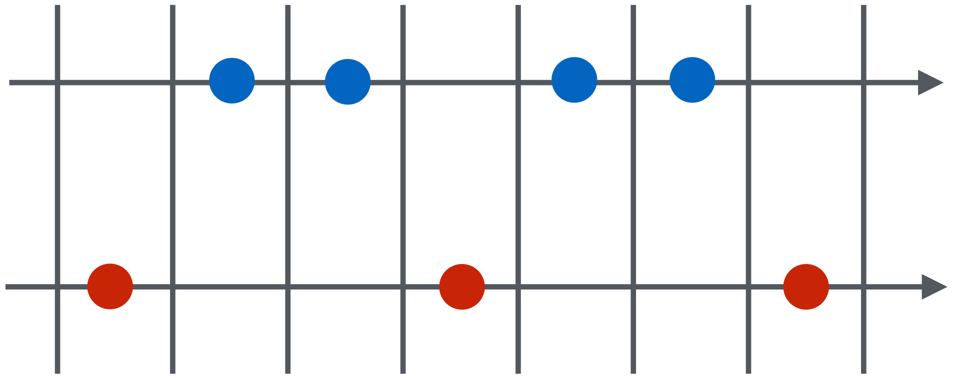
1. Transmissions as unit delays (one step of the logical clock)



Replace transmission with precedence

2. Limit activation interleavings

A process is at most twice as fast as another



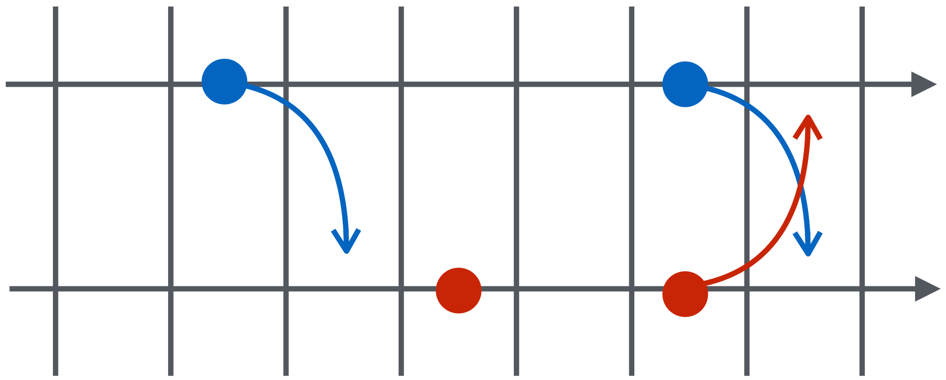
The Quasi-Synchronous Abstraction

Focus on **'almost' synchronous** architectures with **fast transmissions**

Is this abstraction sound?

Reduce the state-space in two ways:

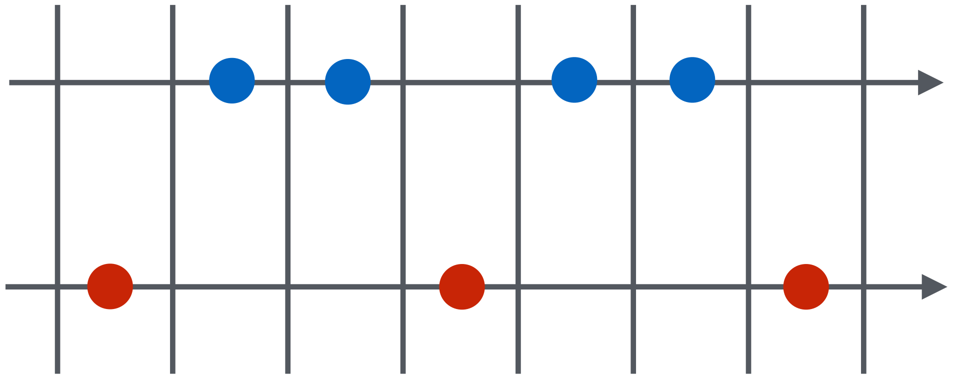
1. Transmissions as unit delays (one step of the logical clock)



Replace transmission with precedence

2. Limit activation interleavings

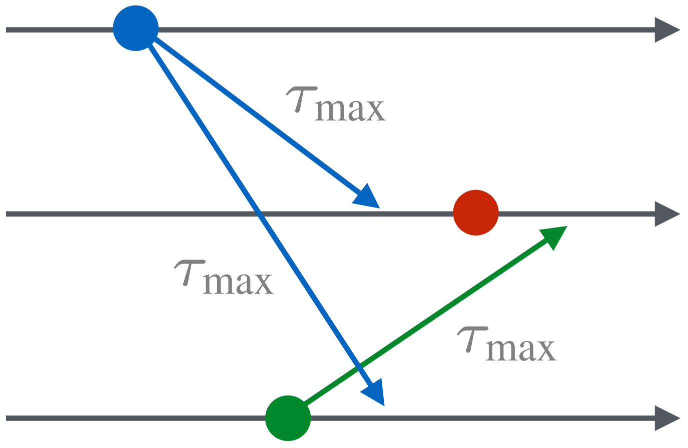
A process is at most twice as fast as another



Unitary Discretization

Definition: A trace is unitary discretizable if there exist a discretization where **transmission** can be modeled as **unit delays**

Theorem: A real-time model with more than two processes is, in general, not unitary discretizable.

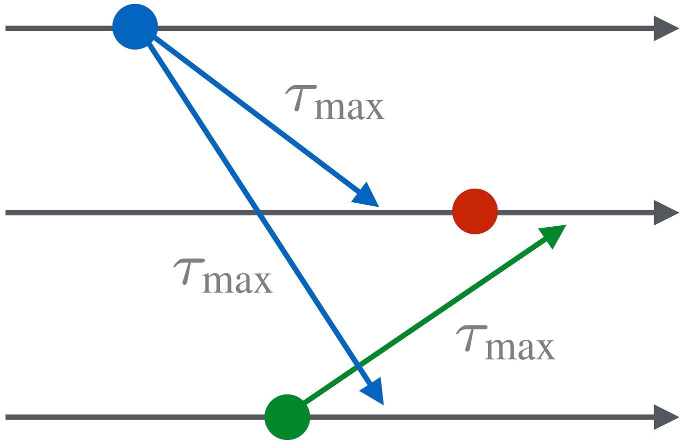


Always possible if transmissions are not instantaneous

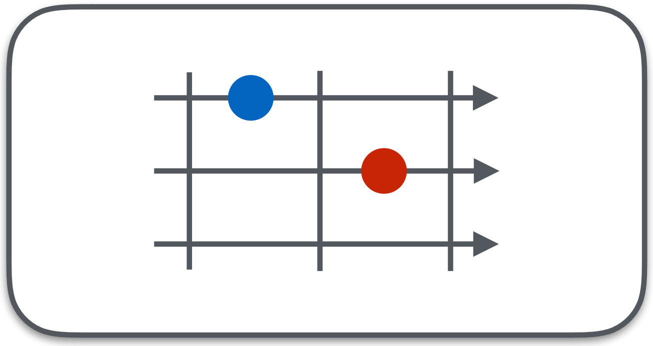
Unitary Discretization

Definition: A trace is unitary discretizable if there exist a discretization where **transmission** can be modeled as **unit delays**

Theorem: A real-time model with more than two processes is, in general, not unitary discretizable.



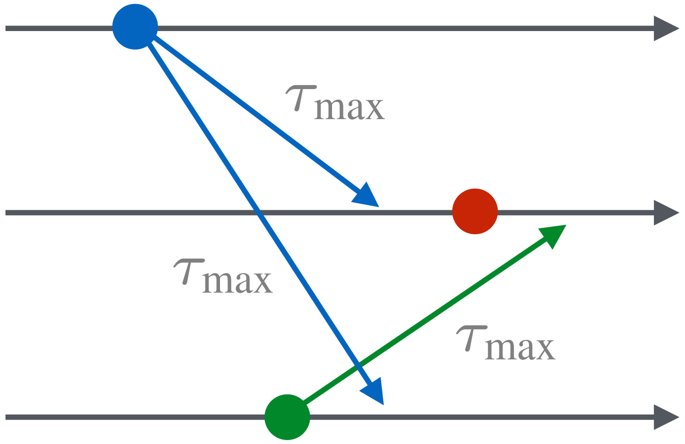
Always possible if transmissions are not instantaneous



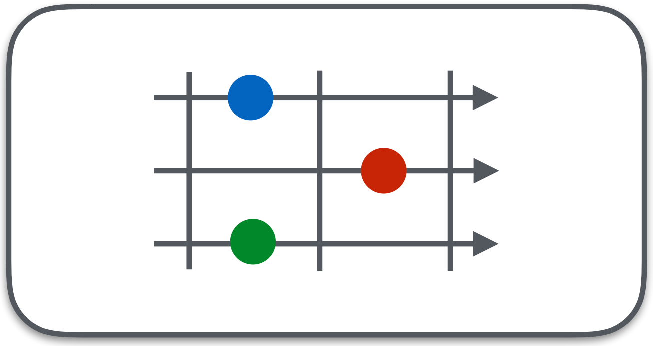
Unitary Discretization

Definition: A trace is unitary discretizable if there exist a discretization where **transmission** can be modeled as **unit delays**

Theorem: A real-time model with more than two processes is, in general, not unitary discretizable.



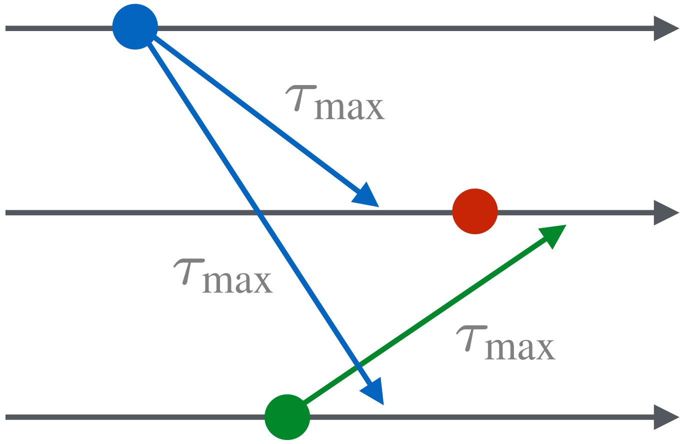
Always possible if transmissions are not instantaneous



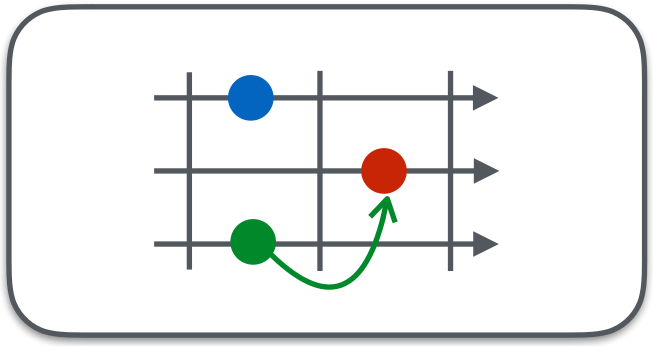
Unitary Discretization

Definition: A trace is unitary discretizable if there exist a discretization where **transmission** can be modeled as **unit delays**

Theorem: A real-time model with more than two processes is, in general, not unitary discretizable.



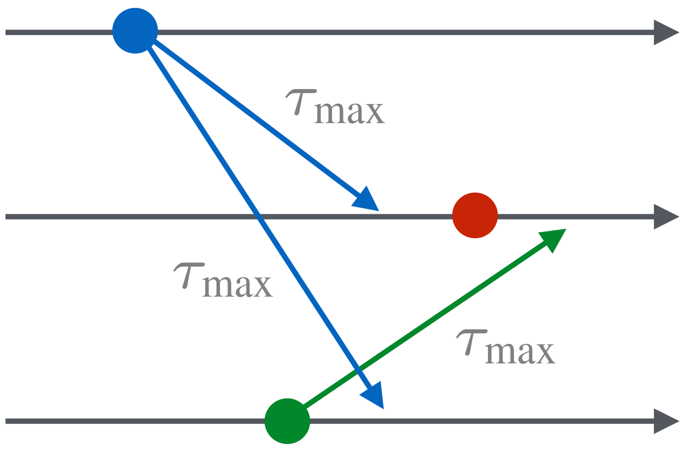
Always possible if transmissions are not instantaneous



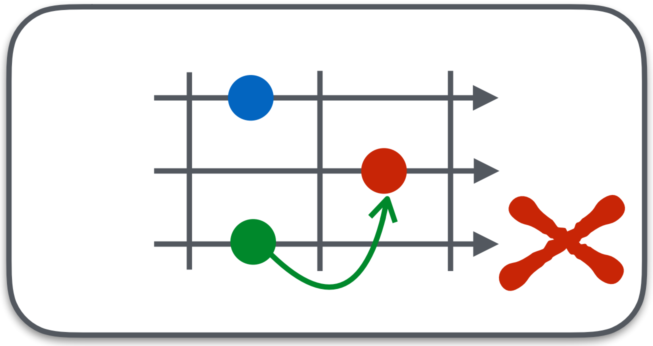
Unitary Discretization

Definition: A trace is unitary discretizable if there exist a discretization where **transmission** can be modeled as **unit delays**

Theorem: A real-time model with more than two processes is, in general, not unitary discretizable.



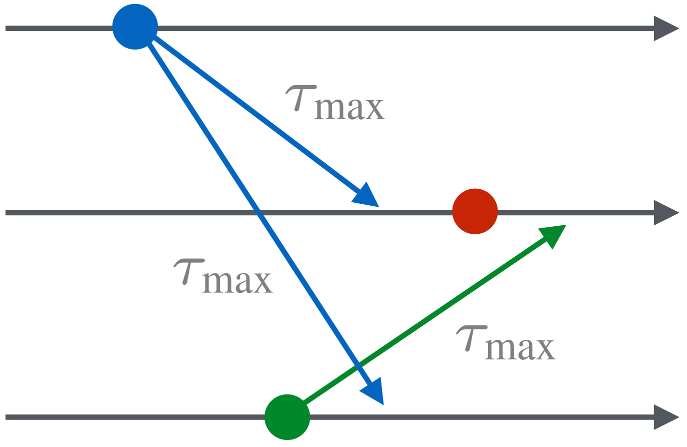
Always possible if transmissions are not instantaneous



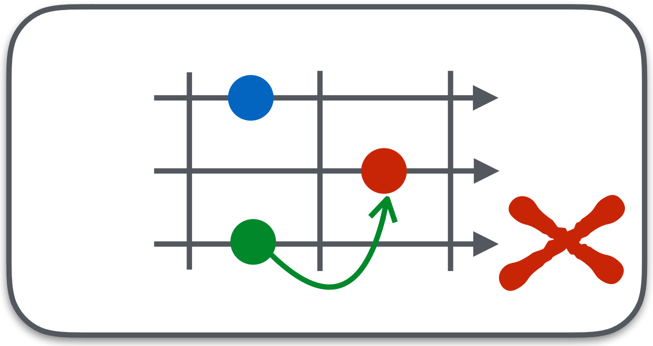
Unitary Discretization

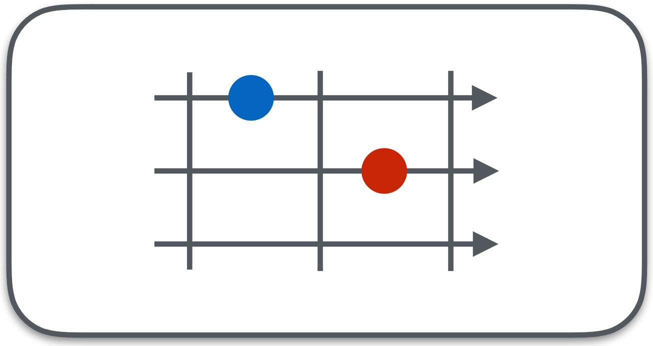
Definition: A trace is unitary discretizable if there exist a discretization where **transmission** can be modeled as **unit delays**

Theorem: A real-time model with more than two processes is, in general, not unitary discretizable.



Always possible if transmissions are not instantaneous

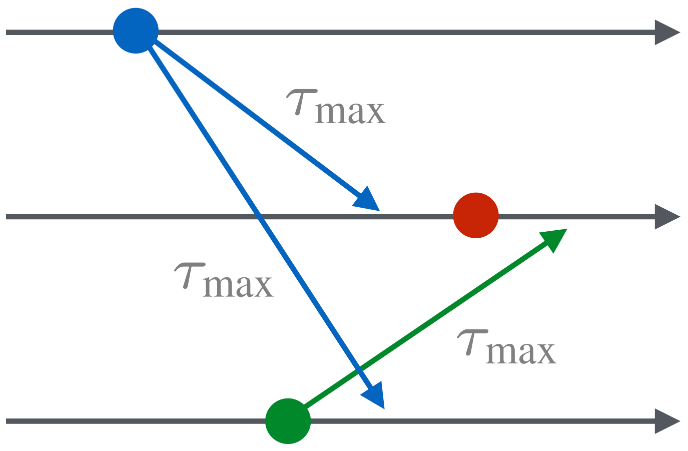




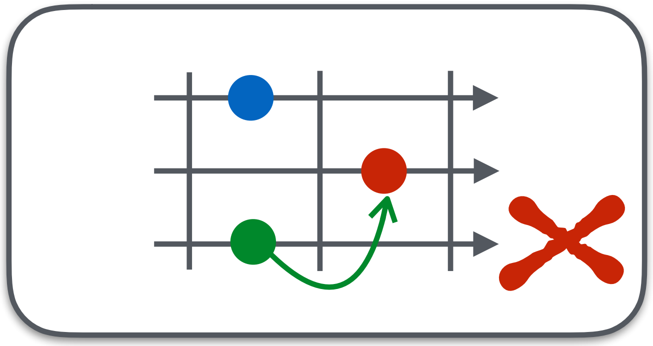
Unitary Discretization

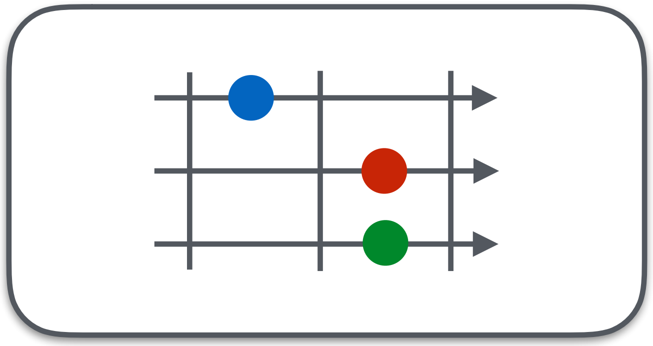
Definition: A trace is unitary discretizable if there exist a discretization where **transmission** can be modeled as **unit delays**

Theorem: A real-time model with more than two processes is, in general, not unitary discretizable.



Always possible if transmissions are not instantaneous

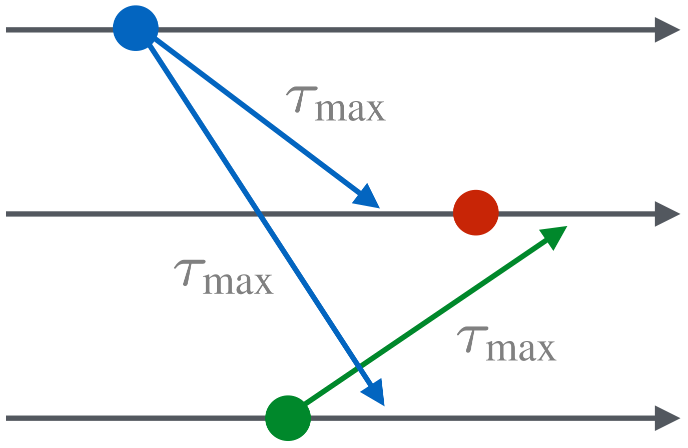




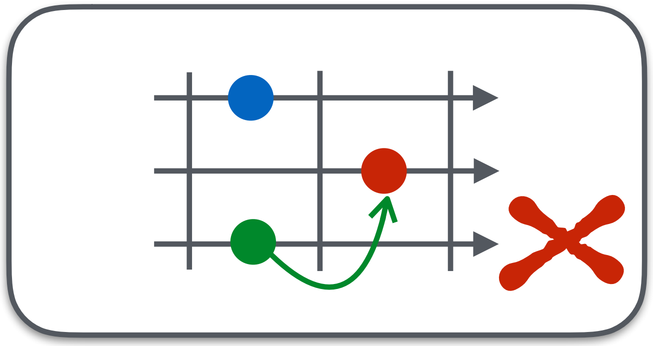
Unitary Discretization

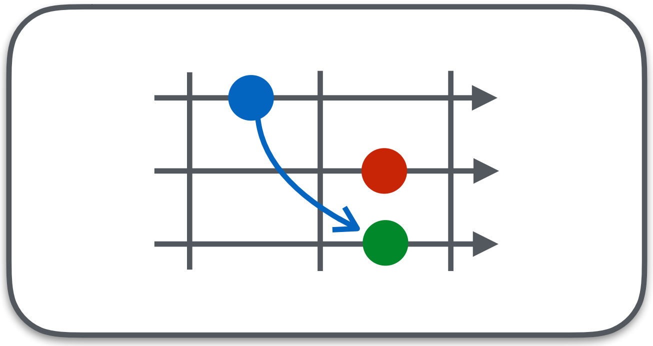
Definition: A trace is unitary discretizable if there exist a discretization where **transmission** can be modeled as **unit delays**

Theorem: A real-time model with more than two processes is, in general, not unitary discretizable.



Always possible if transmissions are not instantaneous

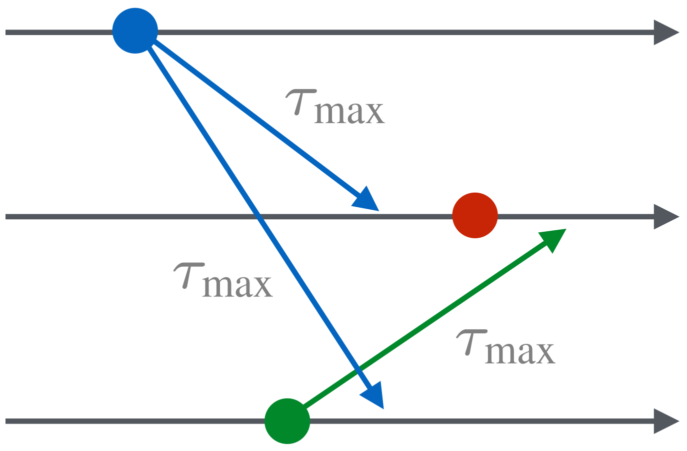




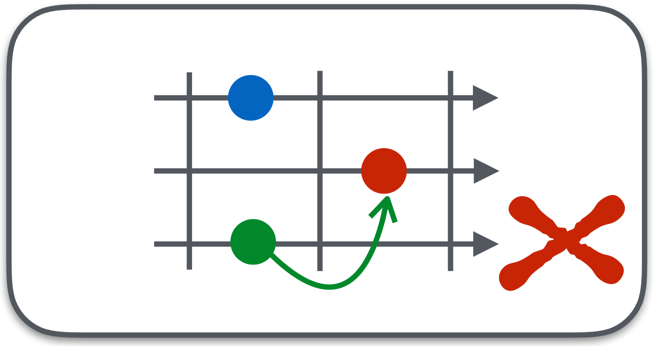
Unitary Discretization

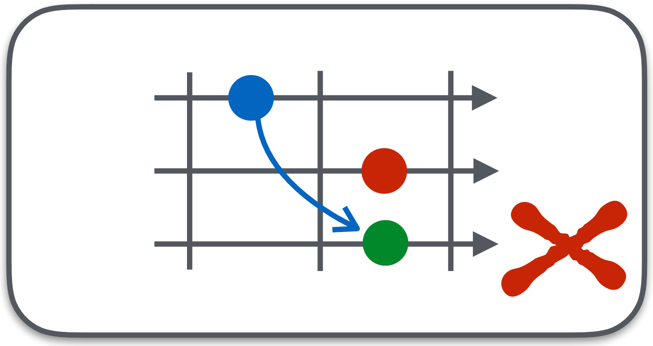
Definition: A trace is unitary discretizable if there exist a discretization where **transmission** can be modeled as **unit delays**

Theorem: A real-time model with more than two processes is, in general, not unitary discretizable.



Always possible if transmissions are not instantaneous

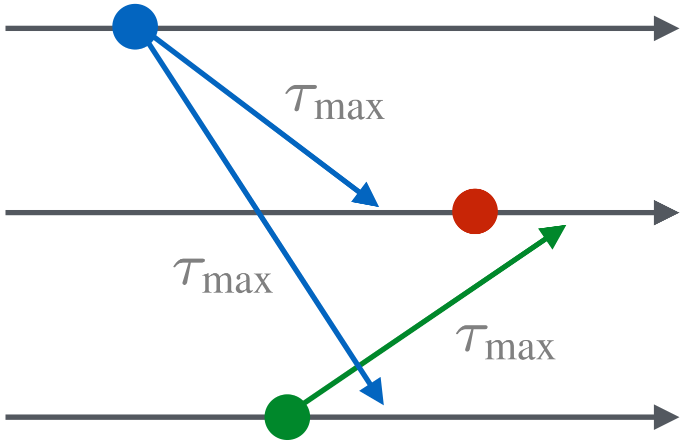




Unitary Discretization

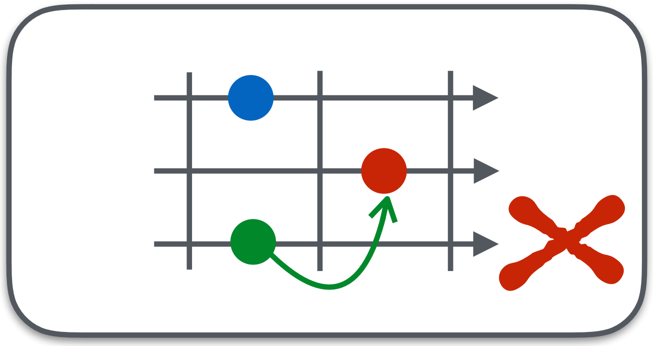
Definition: A trace is unitary discretizable if there exist a discretization where **transmission** can be modeled as **unit delays**

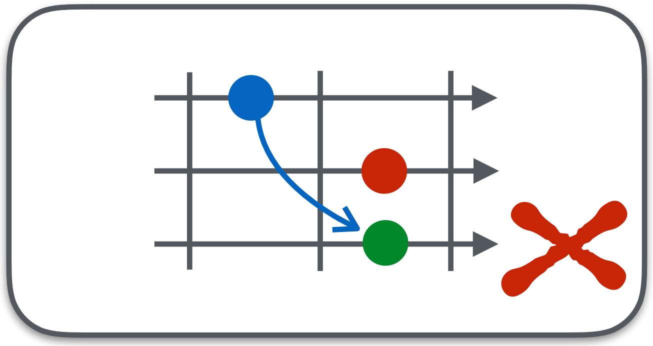
Theorem: A real-time model with more than two processes is, in general, not unitary discretizable.



Some traces are not captured by the discrete abstraction

Always possible if transmissions are not instantaneous



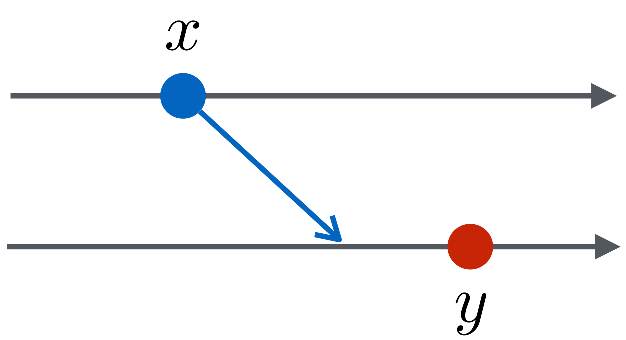


Trace Graph

Gather all contraints on a unitary discretization f in a weighted graph

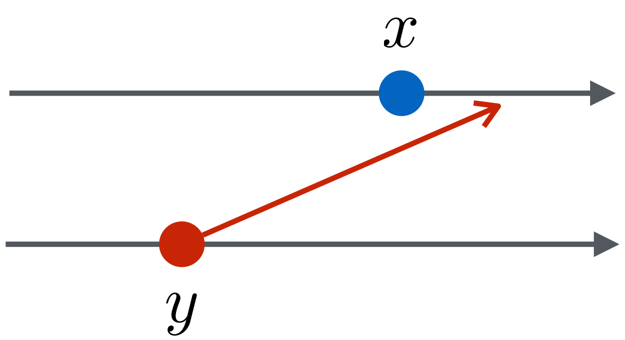
After reception

$$x \xrightarrow{1} y \implies f(x) < f(y)$$



Before reception

$$x \xrightarrow{0} y \implies f(x) \leq f(y)$$

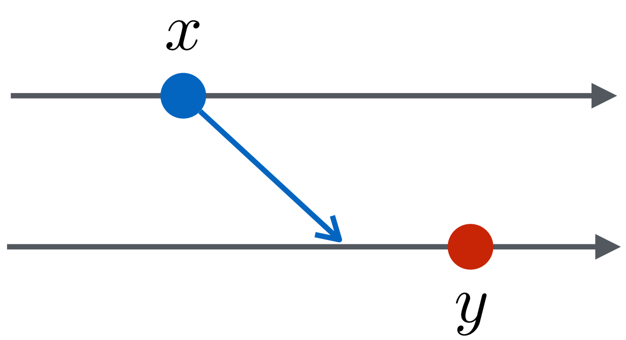


Trace Graph

Gather all contraints on a unitary discretization f in a weighted graph

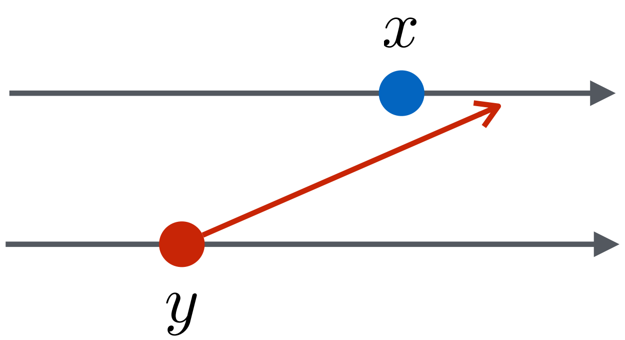
After reception

$$x \xrightarrow{1} y \implies f(x) < f(y)$$



Before reception

$$x \xrightarrow{0} y \implies f(x) \leq f(y)$$



Lemma: A trace is unitary discretizable if and only if there are no cycle of positive weight in the associated trace graph.

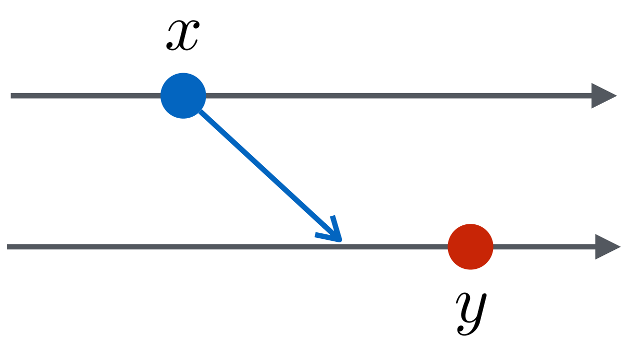
Definition: A real-time model is *unitary discretizable* if all possible traces are unitary discretizable.

Trace Graph

Gather all contraints on a unitary discretization f in a weighted graph

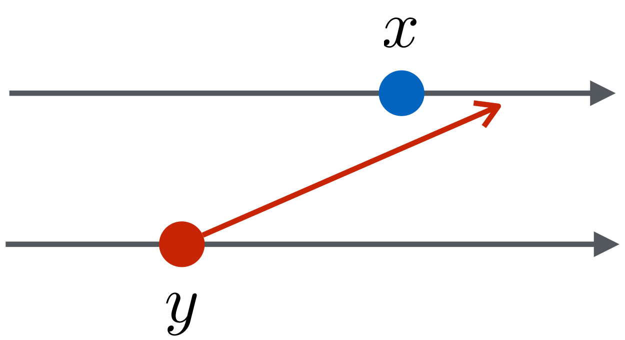
After reception

$$x \xrightarrow{1} y \implies f(x) < f(y)$$

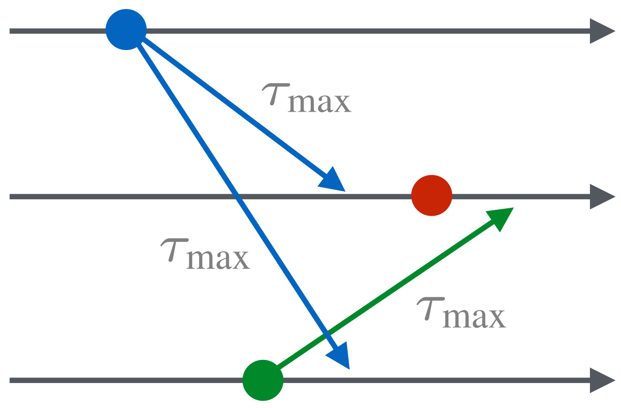


Before reception

$$x \xrightarrow{0} y \implies f(x) \leq f(y)$$



Lemma: A trace is unitary discretizable if and only if there are no cycle of positive weight in the associated trace graph.



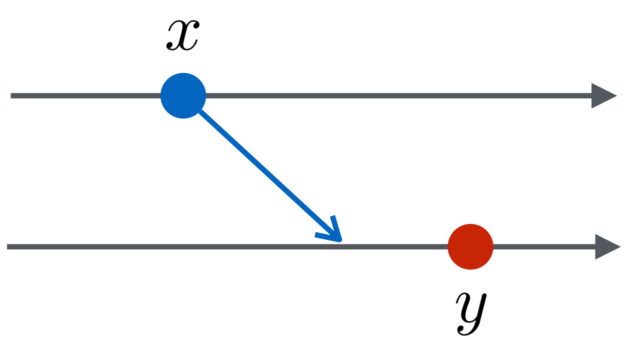
Definition: A real-time model is *unitary discretizable* if all possible traces are unitary discretizable.

Trace Graph

Gather all contraints on a unitary discretization f in a weighted graph

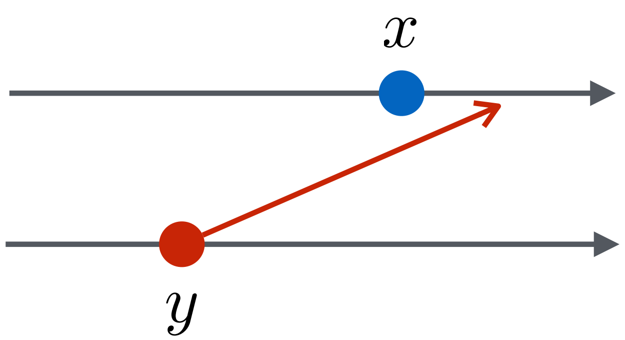
After reception

$$x \xrightarrow{1} y \implies f(x) < f(y)$$

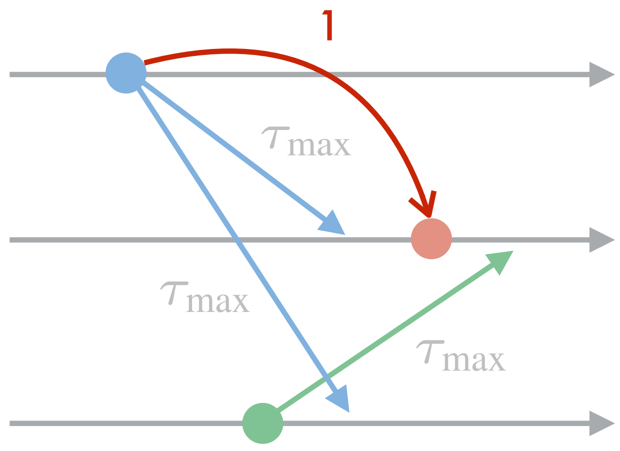


Before reception

$$x \xrightarrow{0} y \implies f(x) \leq f(y)$$



Lemma: A trace is unitary discretizable if and only if there are no cycle of positive weight in the associated trace graph.



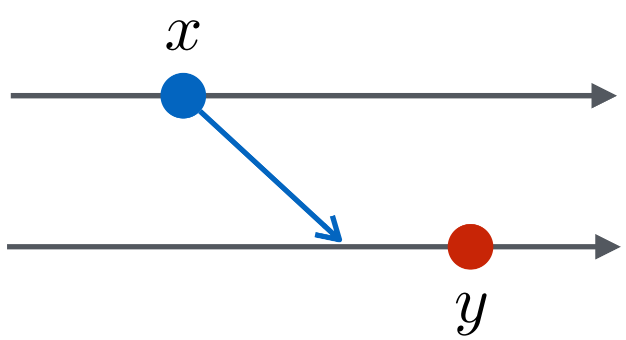
Definition: A real-time model is *unitary discretizable* if all possible traces are unitary discretizable.

Trace Graph

Gather all contraints on a unitary discretization f in a weighted graph

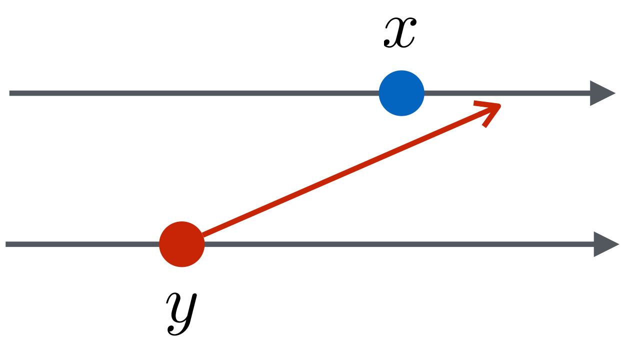
After reception

$x \xrightarrow{1} y \implies f(x) < f(y)$

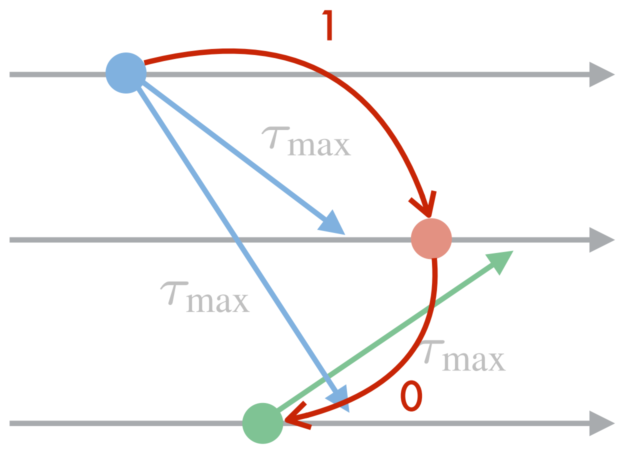


Before reception

$x \xrightarrow{0} y \implies f(x) \leq f(y)$



Lemma: A trace is unitary discretizable if and only if there are no cycle of positive weight in the associated trace graph.



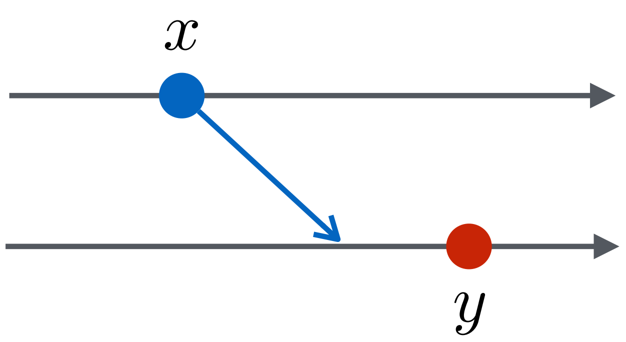
Definition: A real-time model is *unitary discretizable* if all possible traces are unitary discretizable.

Trace Graph

Gather all contraints on a unitary discretization f in a weighted graph

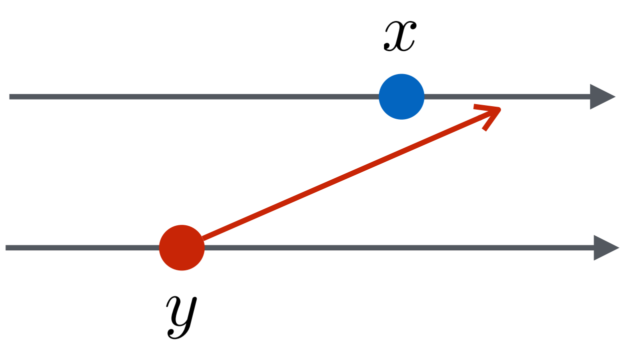
After reception

$$x \xrightarrow{1} y \implies f(x) < f(y)$$

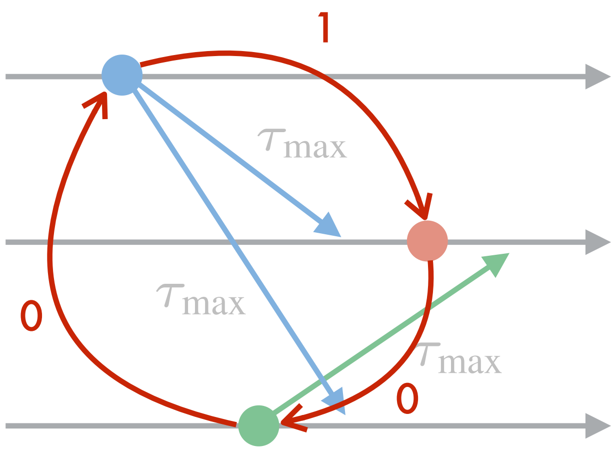


Before reception

$$x \xrightarrow{0} y \implies f(x) \leq f(y)$$



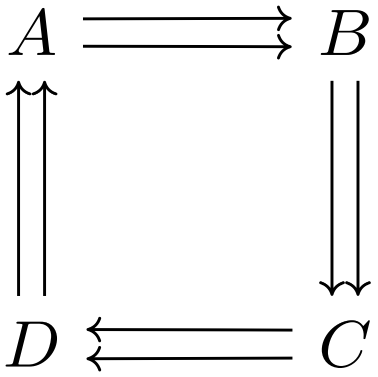
Lemma: A trace is unitary discretizable if and only if there are no cycle of positive weight in the associated trace graph.



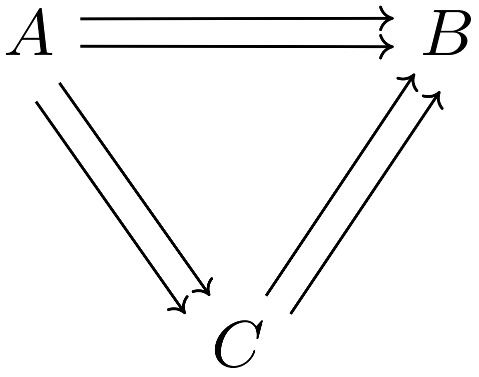
Definition: A real-time model is *unitary discretizable* if all possible traces are unitary discretizable.

Recovering Soundness

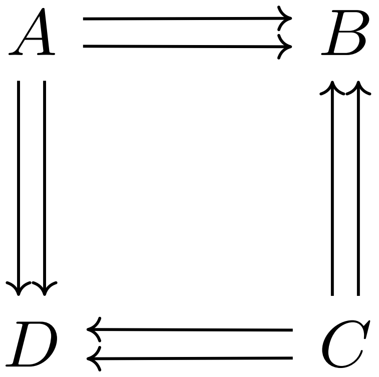
Forbidden topologies in the static communication graph



cycle



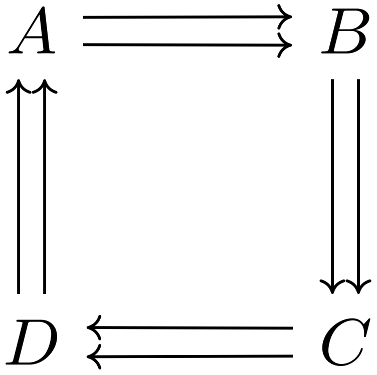
u-cycle



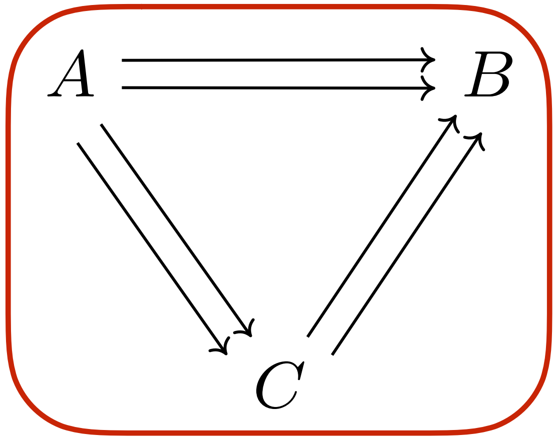
balanced u-cycle

Recovering Soundness

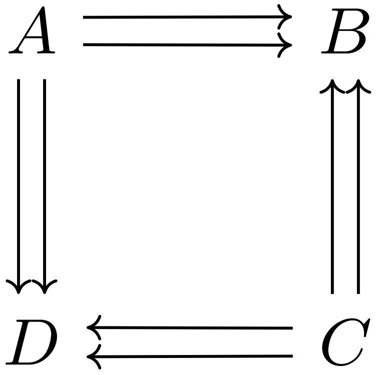
Forbidden topologies in the static communication graph



cycle



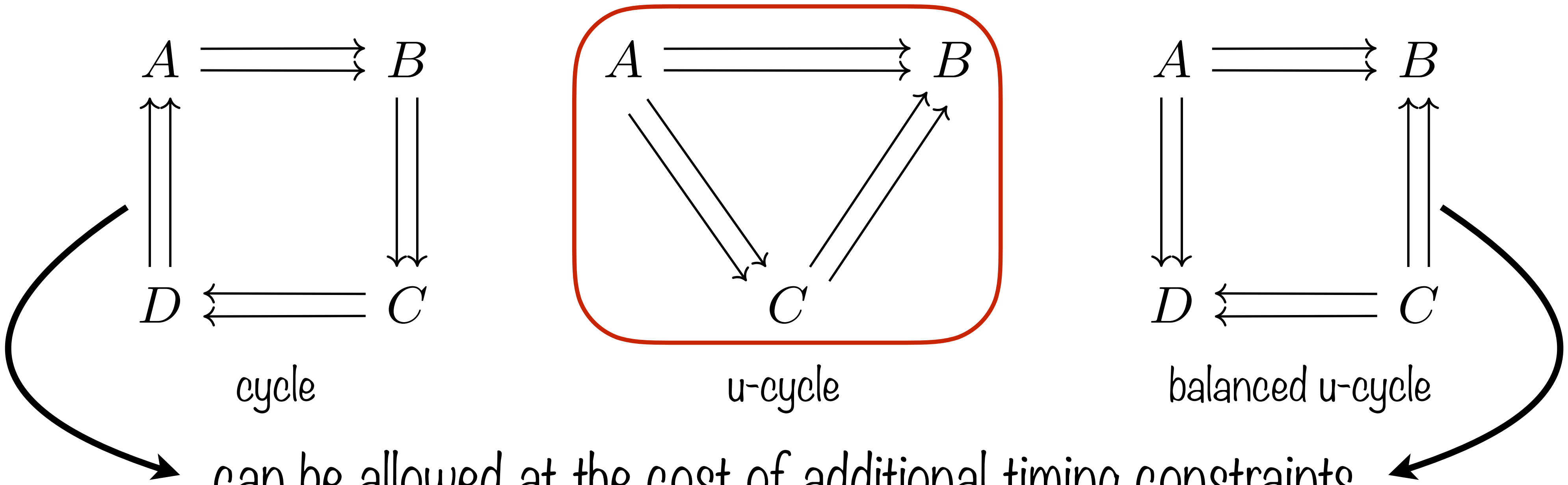
u-cycle



balanced u-cycle

Recovering Soundness

Forbidden topologies in the static communication graph



cycle

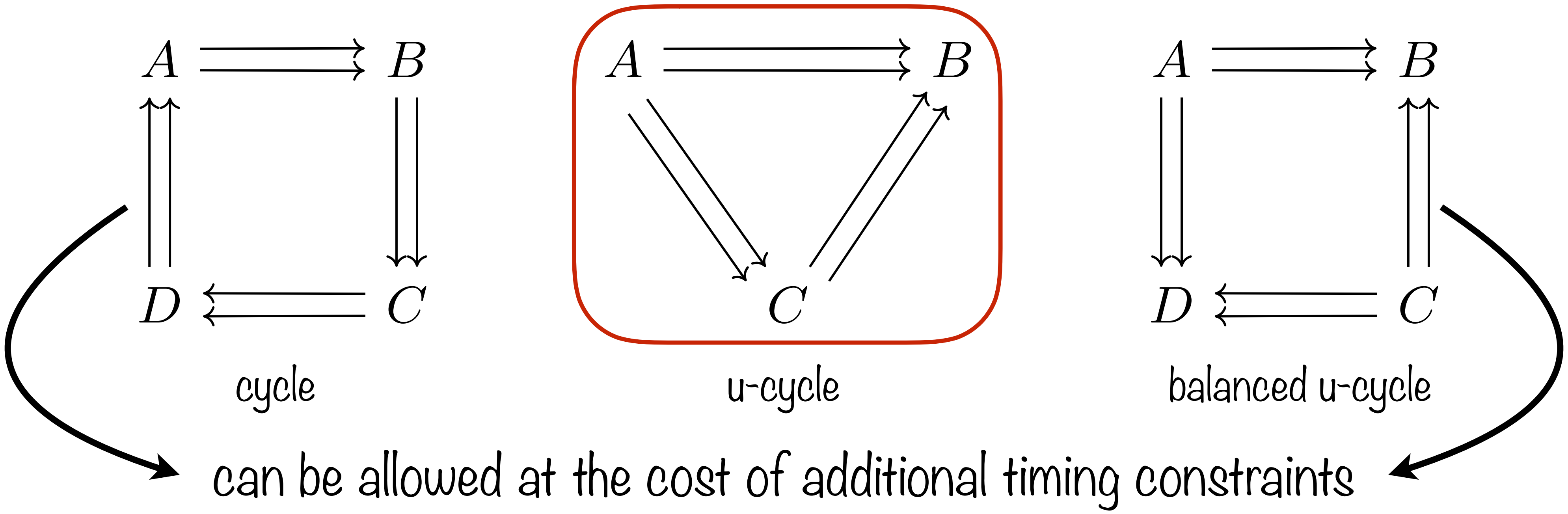
u-cycle

balanced u-cycle

can be allowed at the cost of additional timing constraints

Recovering Soundness

Forbidden topologies in the static communication graph



cycle

u-cycle

balanced u-cycle

can be allowed at the cost of additional timing constraints

Theorem: A quasi-periodic architecture is unitary discretizable if and only if, in the communication graph

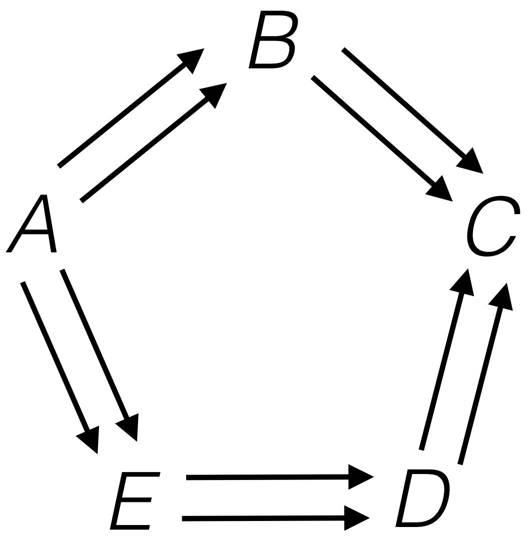
1. All u-cycles are cycles of balanced u-cycle, or $\tau_{\max} = 0$, and
2. There is no balanced u-cycle, or $\tau_{\min} = \tau_{\max}$, and
3. There is no cycle in the communication graph, or $T_{\min} \geq L_c \tau_{\max}$

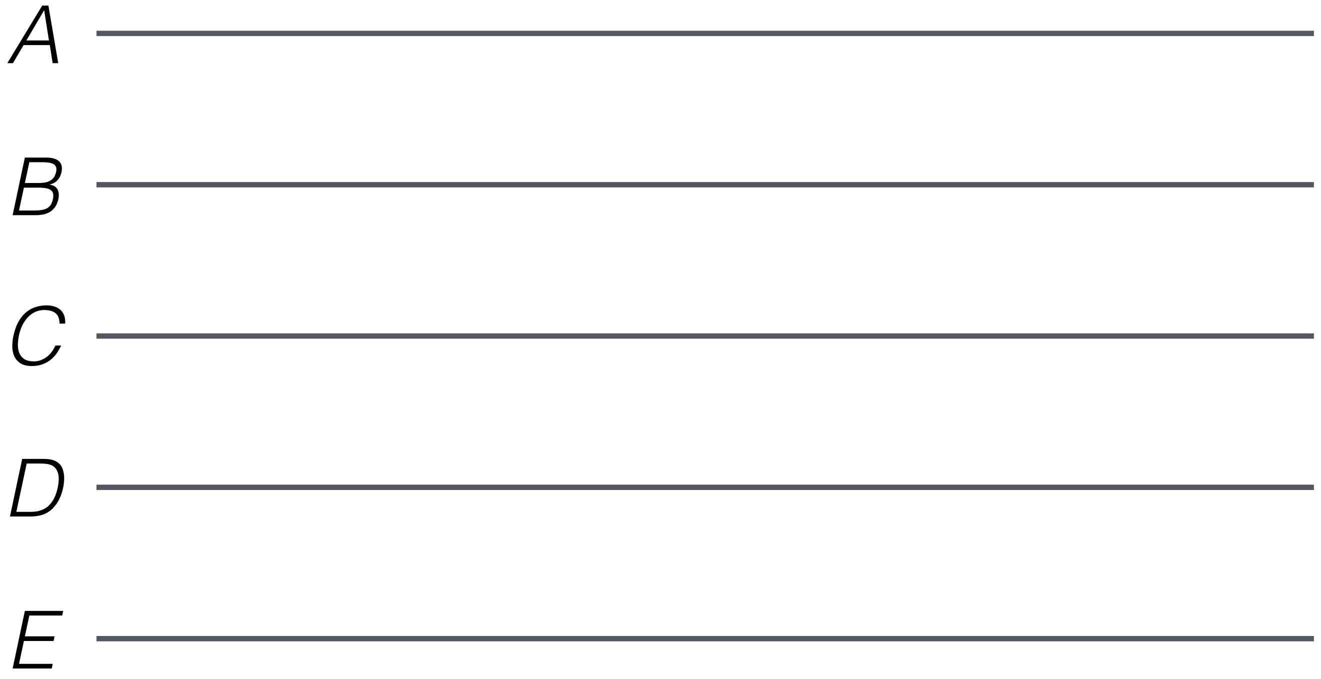
L_C: size of the longest elementary cycle

Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

Communications

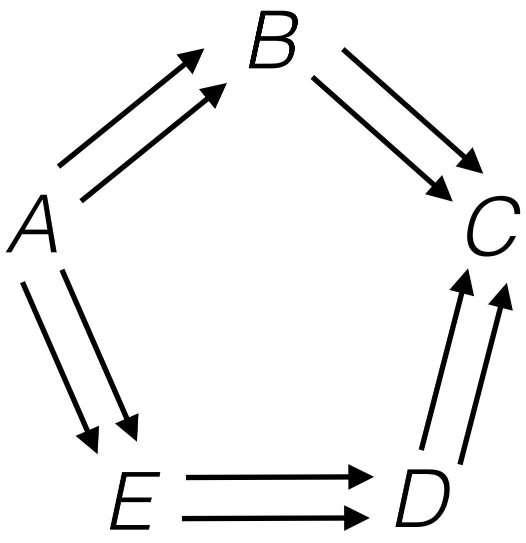


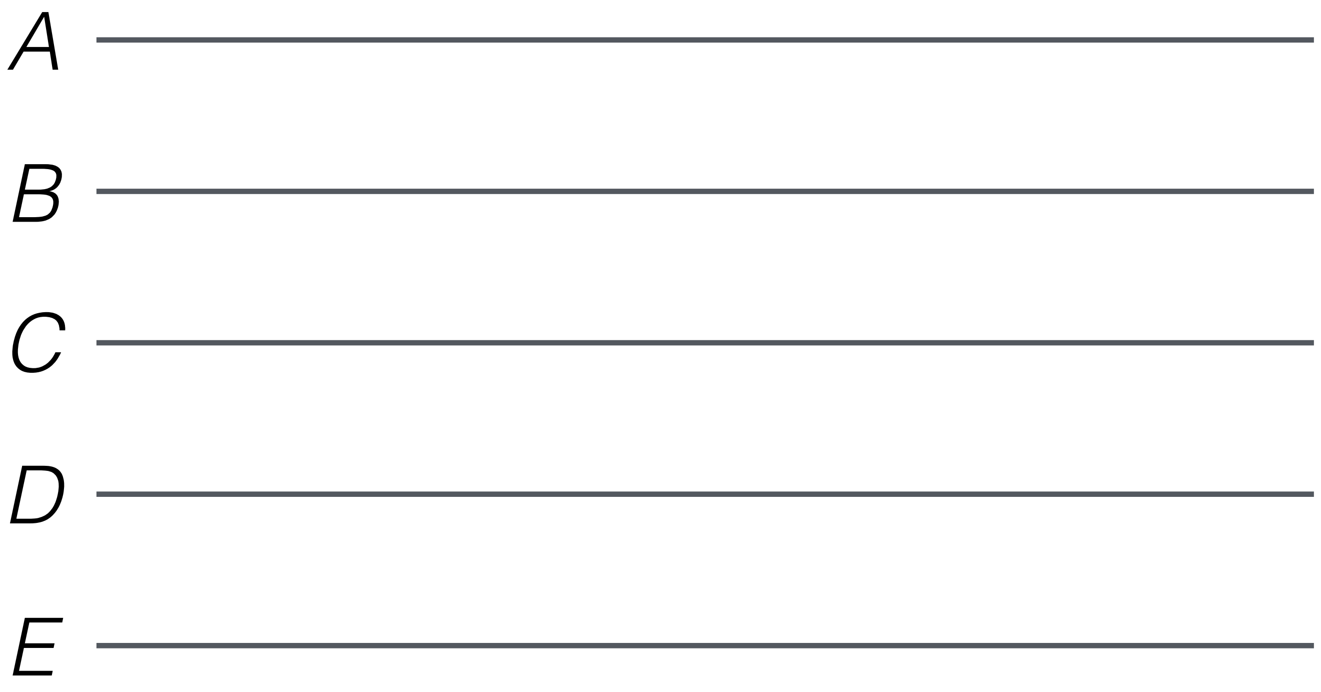


Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

Communications





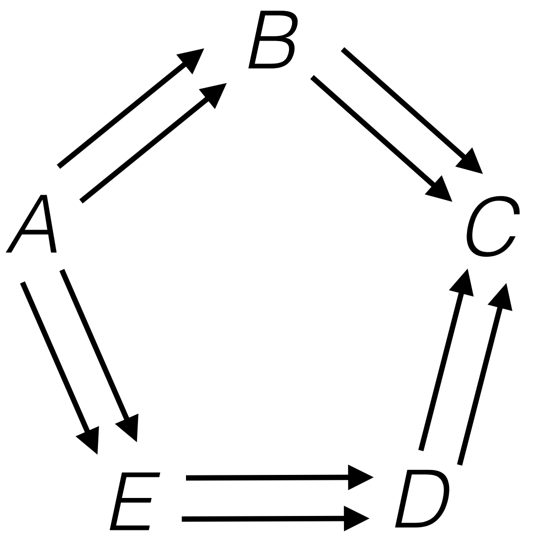
q = 3: # ⇇

p = 2: # ⇉

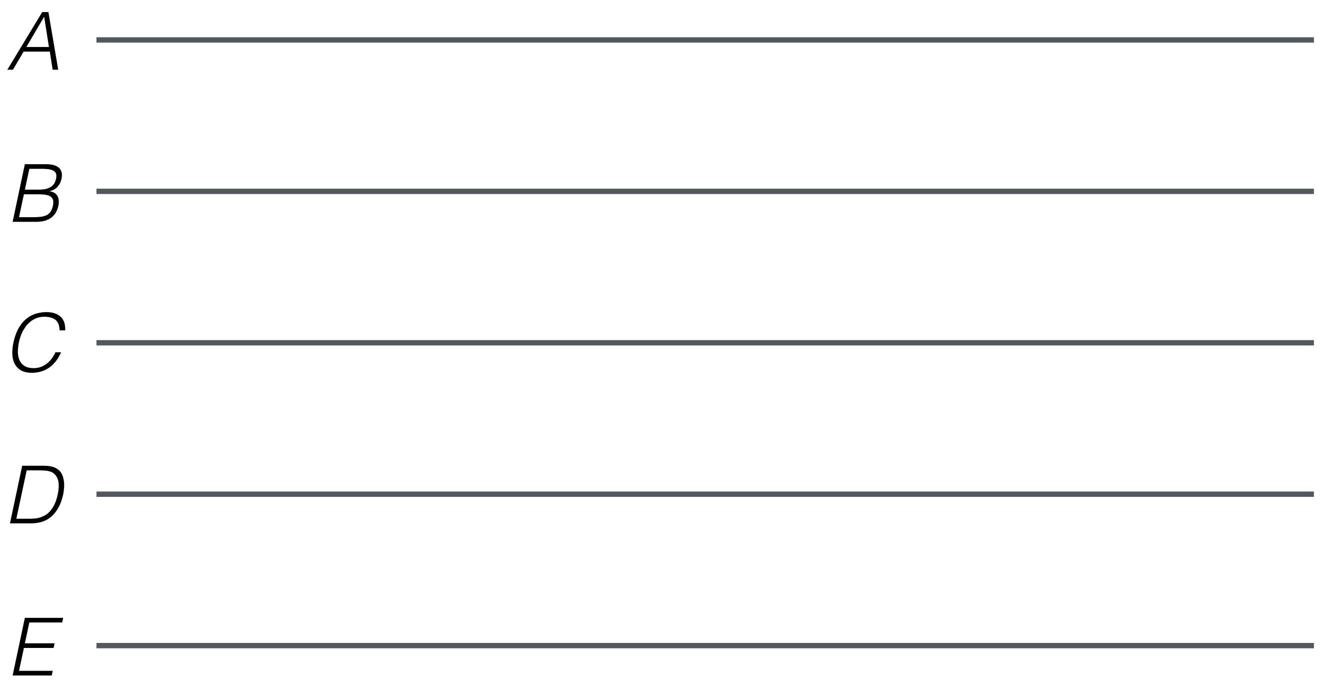
Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

Communications



$q > p \Longrightarrow \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$



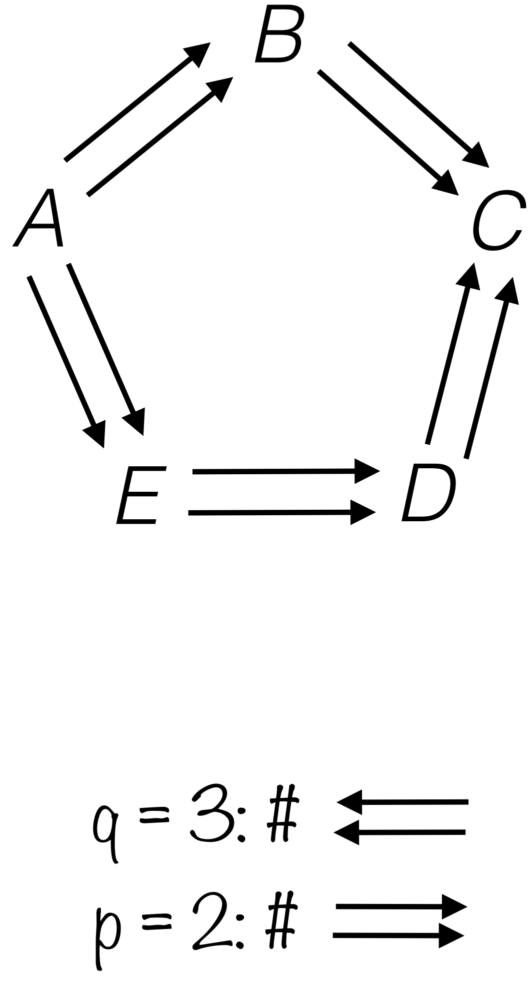
q = 3: # $\leftleftarrows$

p = 2: # $\rightrightarrows$

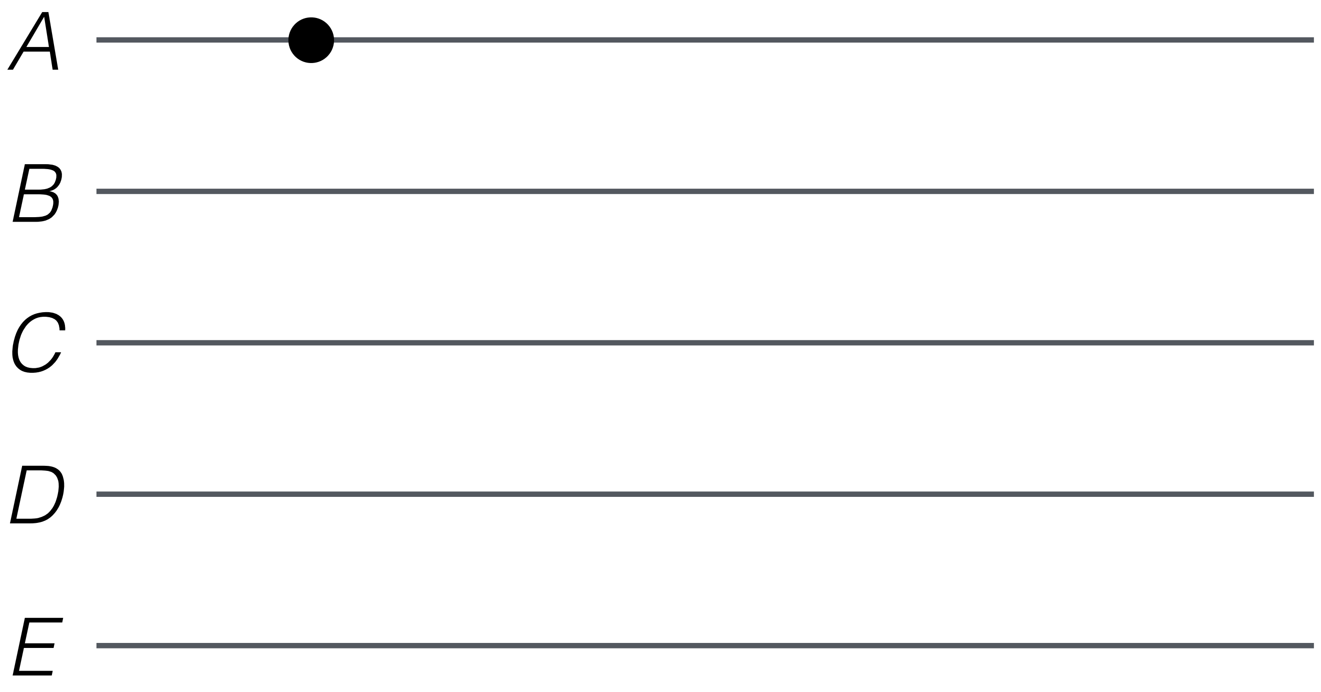
Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

Communications



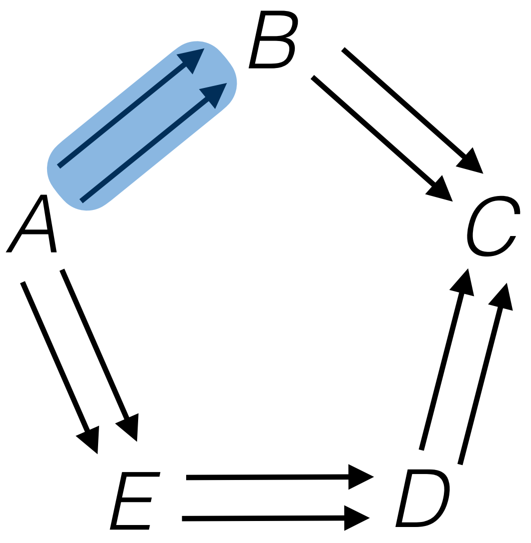
$q > p \Longrightarrow \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$



Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

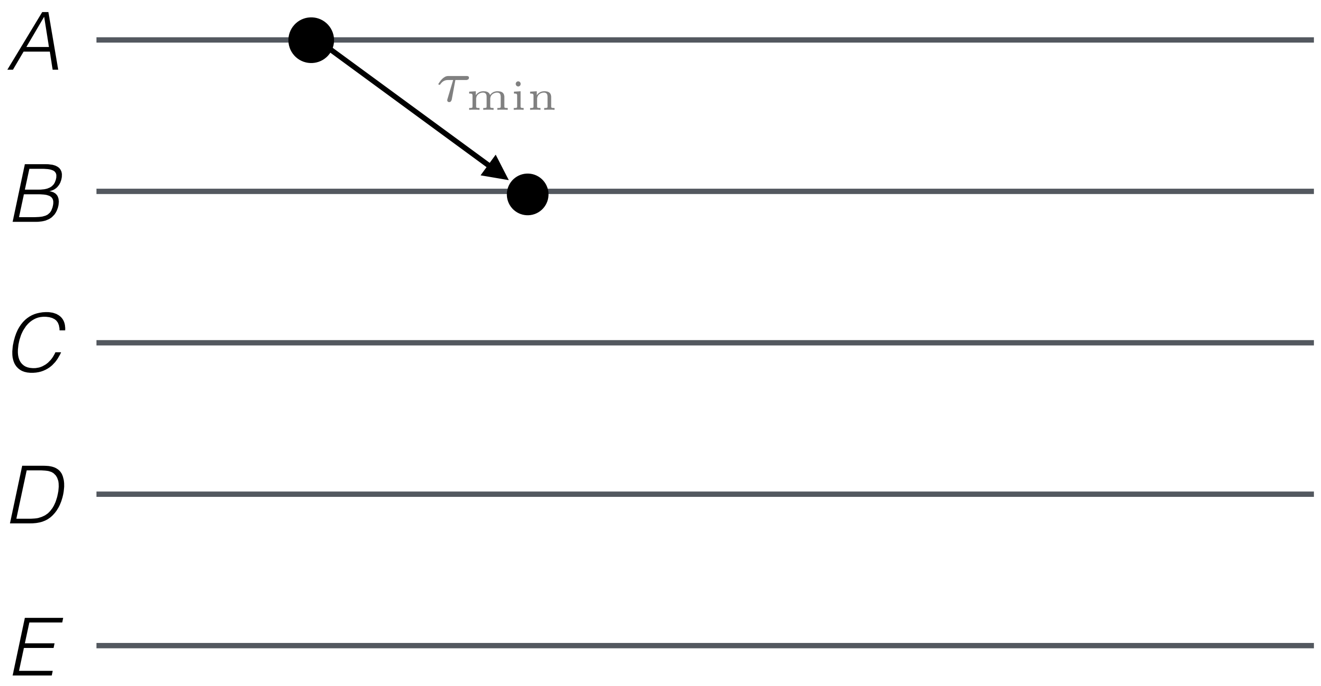
Communications



q = 3: # $\leftleftarrows$

p = 2: # $\rightrightarrows$

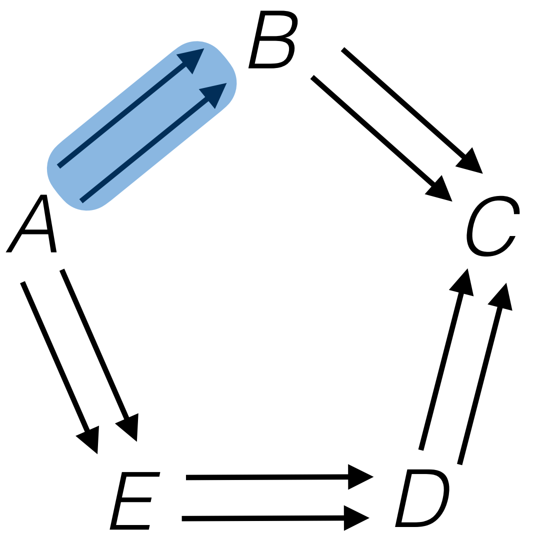
$$q > p \implies \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$$



Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

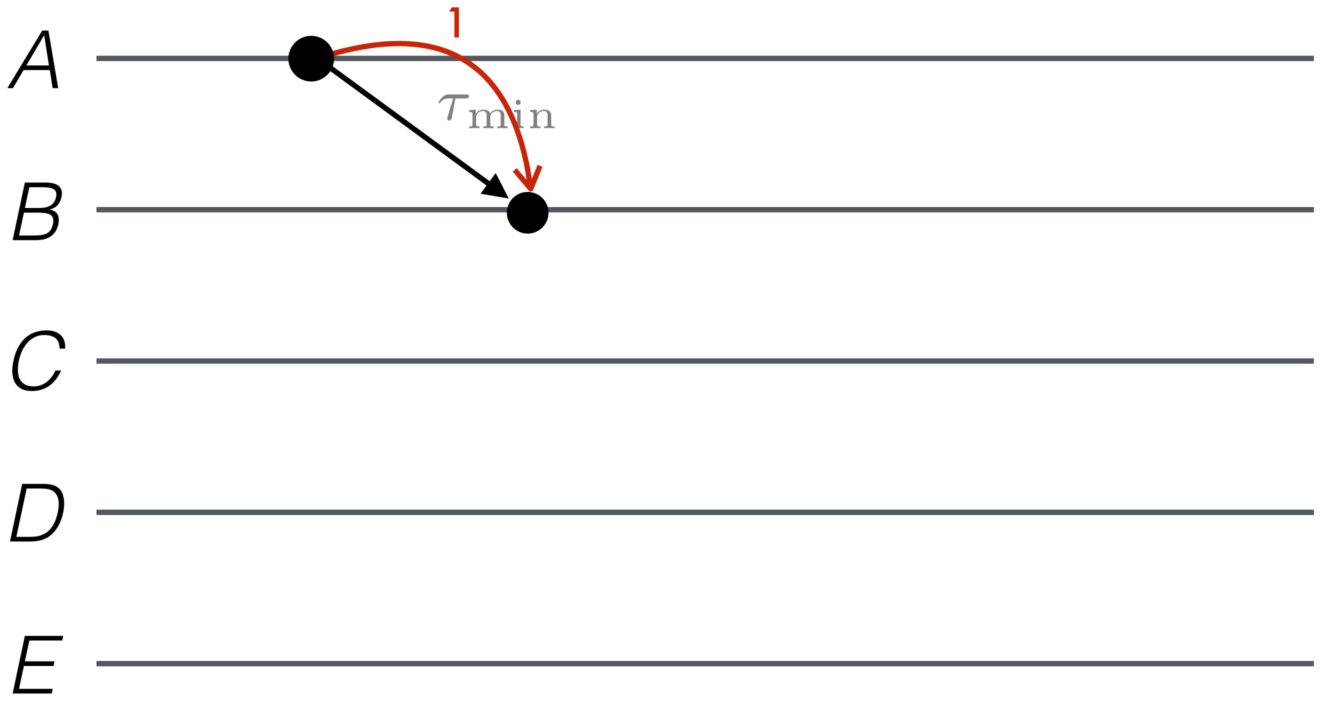
Communications



q = 3: # ⇇

p = 2: # ⇉

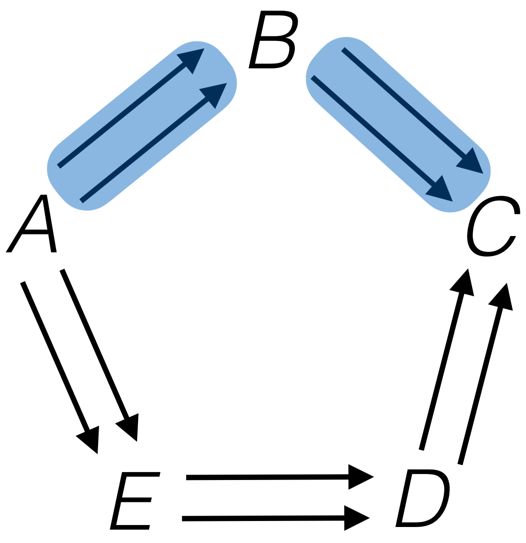
$$q > p \implies \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$$



Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

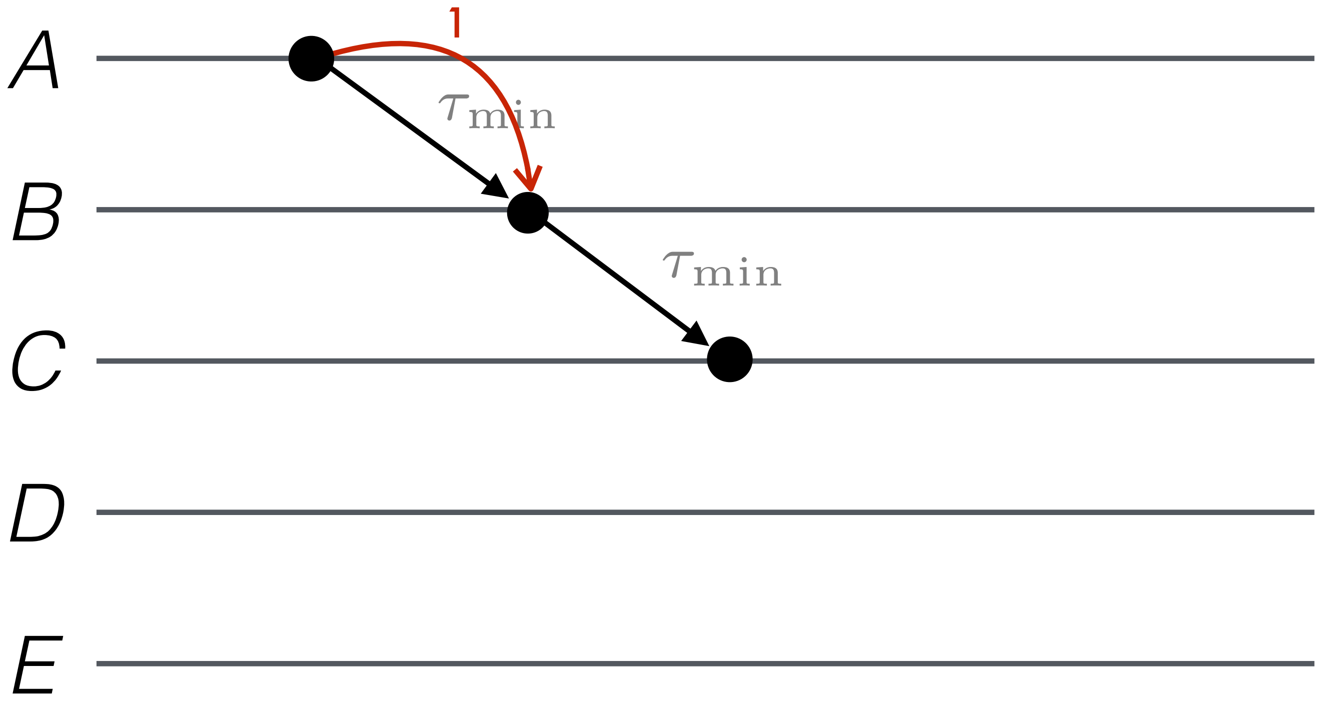
Communications



q = 3: # ⇇

p = 2: # ⇉

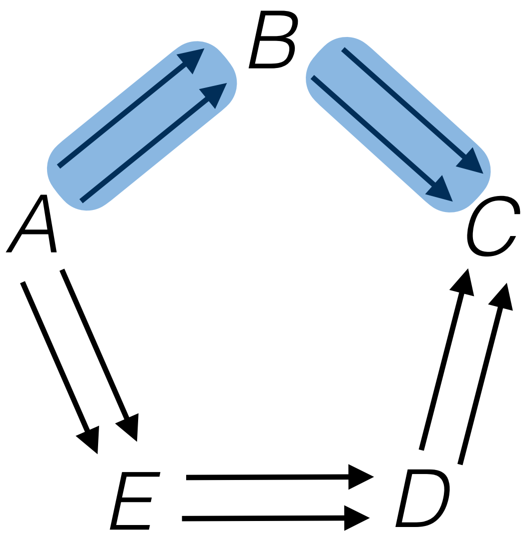
$$q > p \implies \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$$



Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

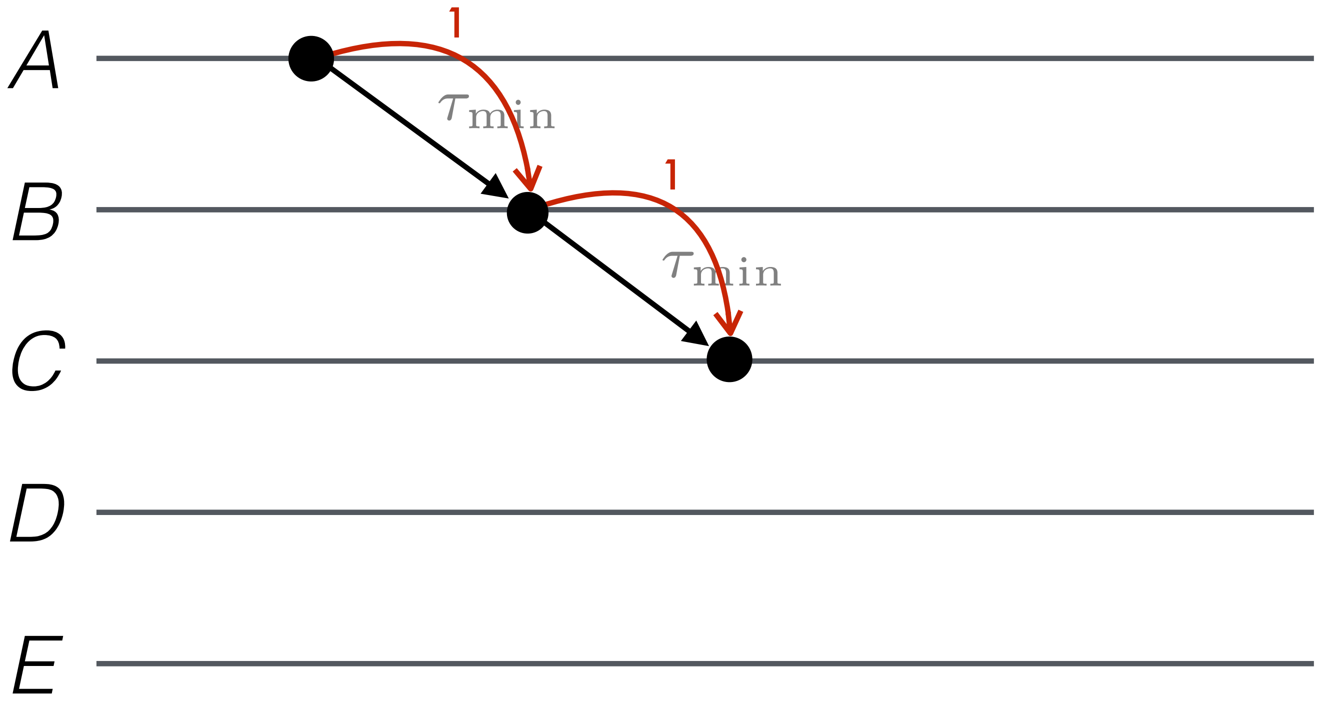
Communications



$q = 3$: # ⇇

$p = 2$: # ⇉

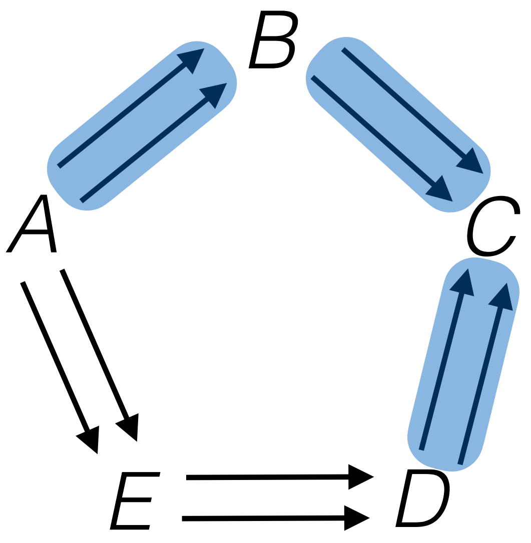
$$q > p \Longrightarrow \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$$



Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

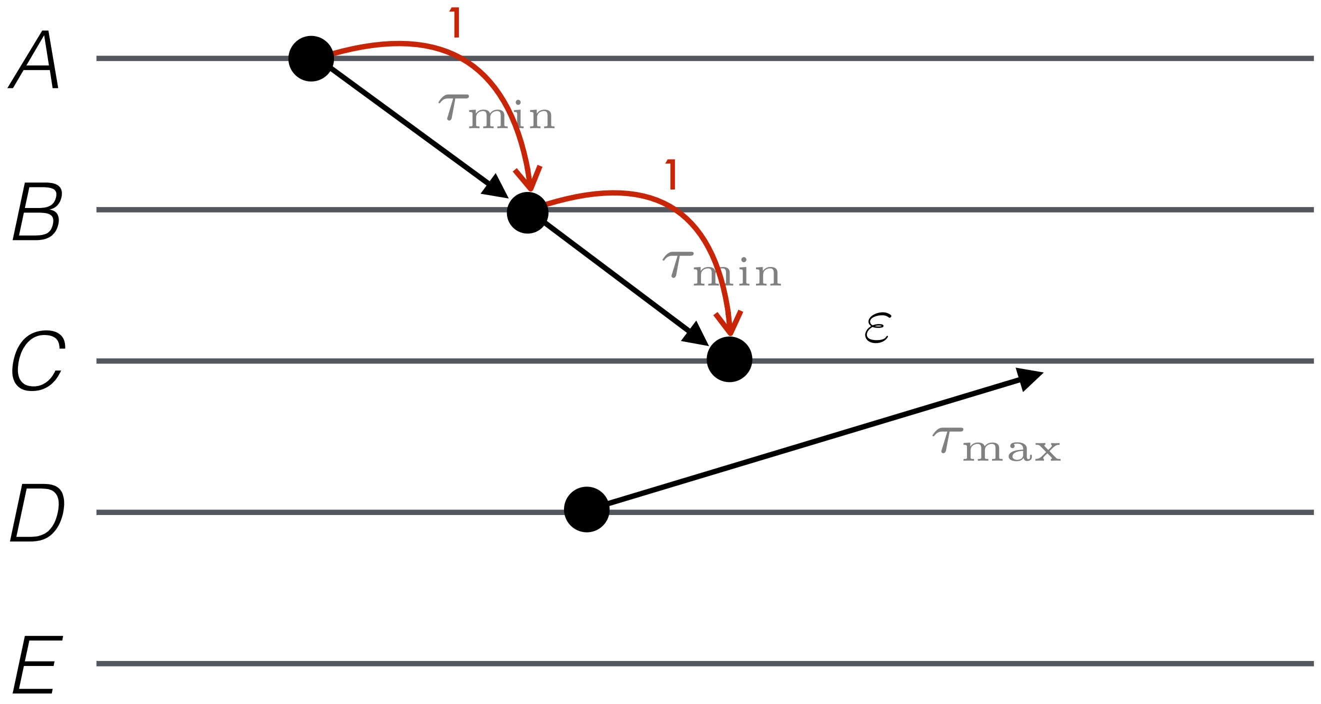
Communications



q = 3: # ⇇

p = 2: # ⇉

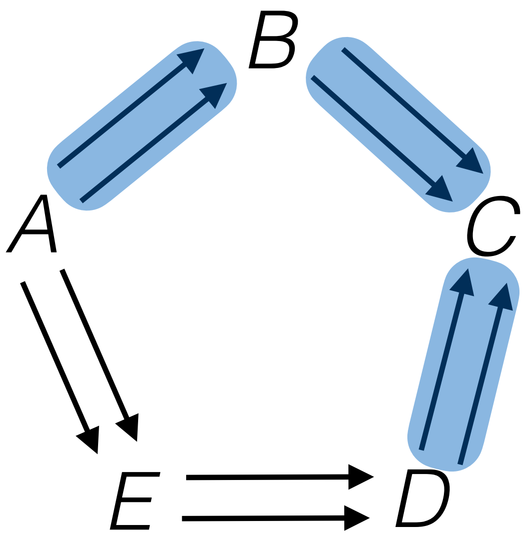
$$q > p \implies \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$$



Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

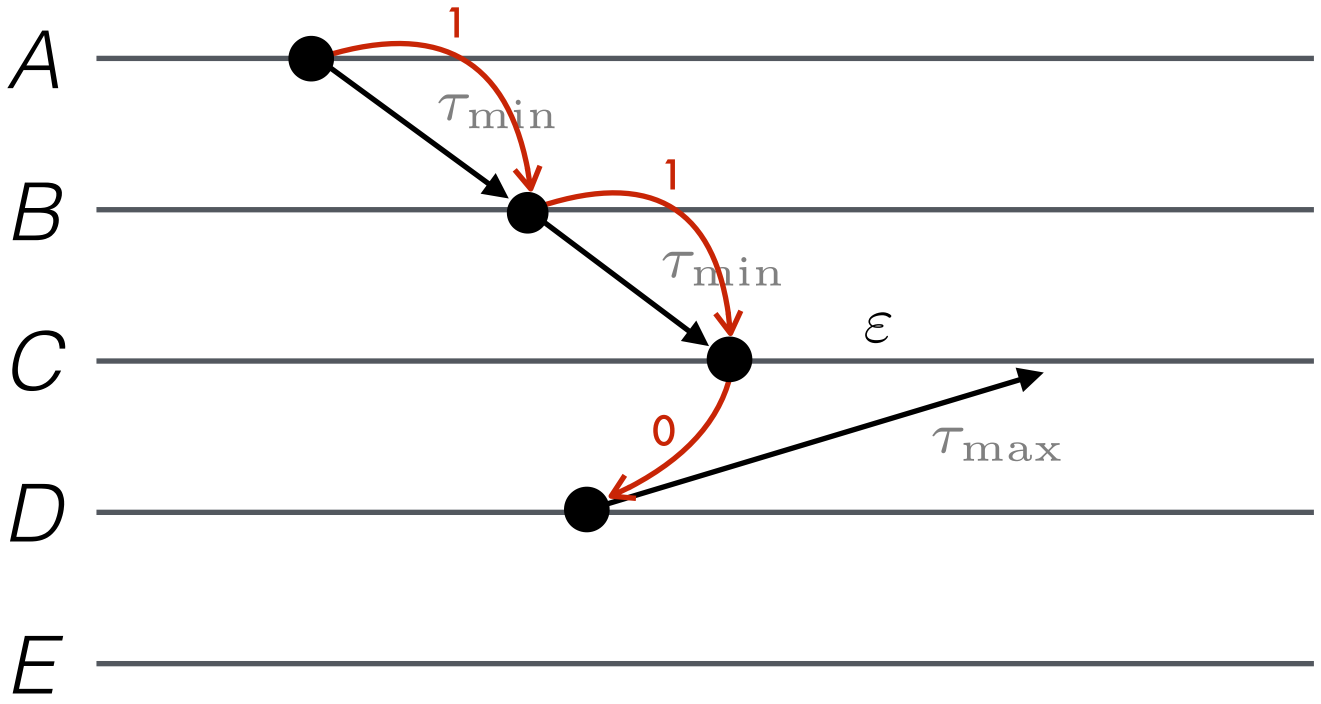
Communications



$q = 3$: # ⇇

$p = 2$: # ⇉

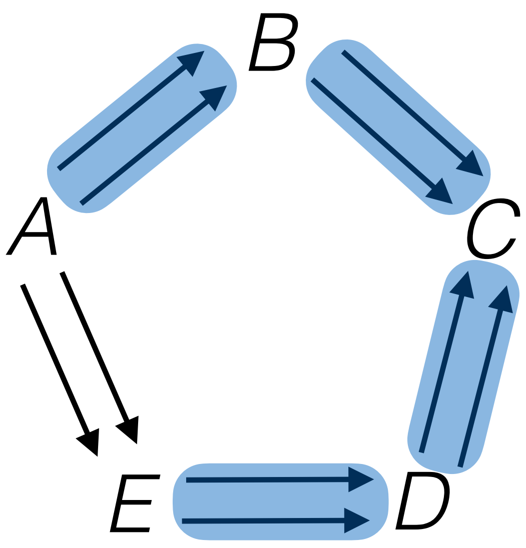
$$q > p \implies \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$$



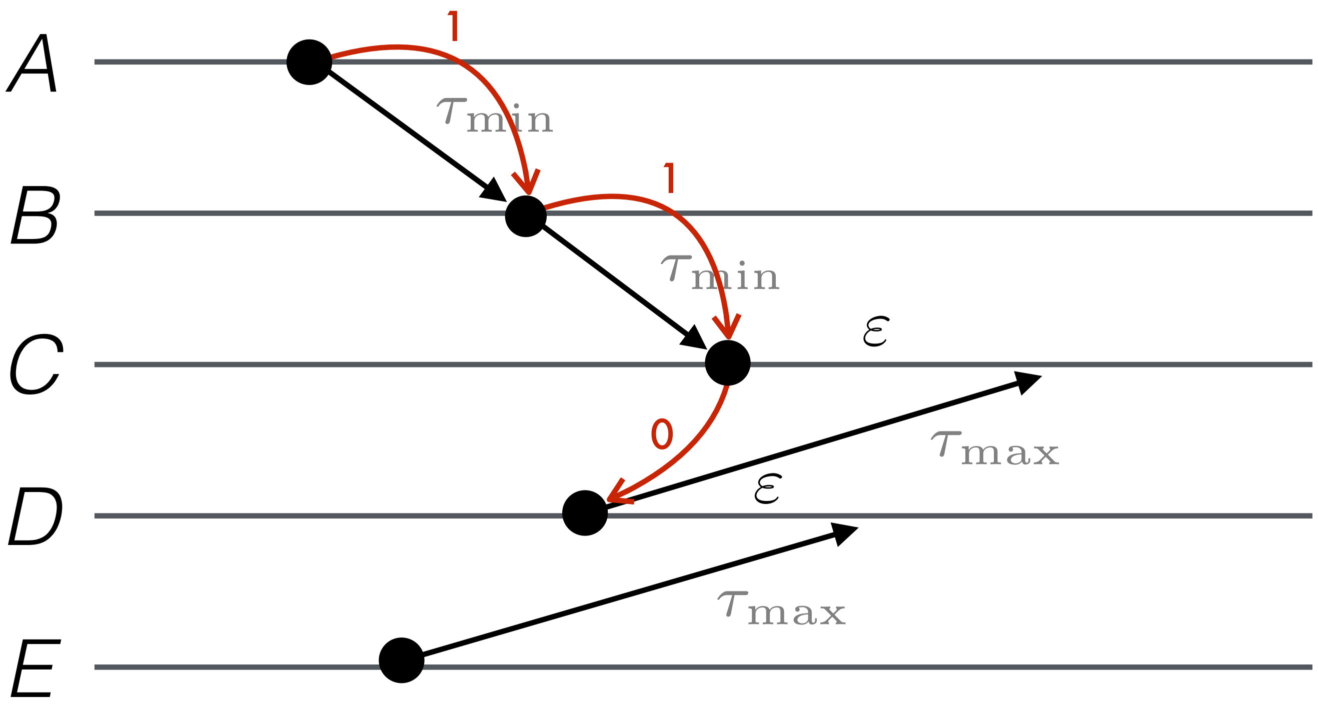
Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

Communications



$$q > p \implies \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$$



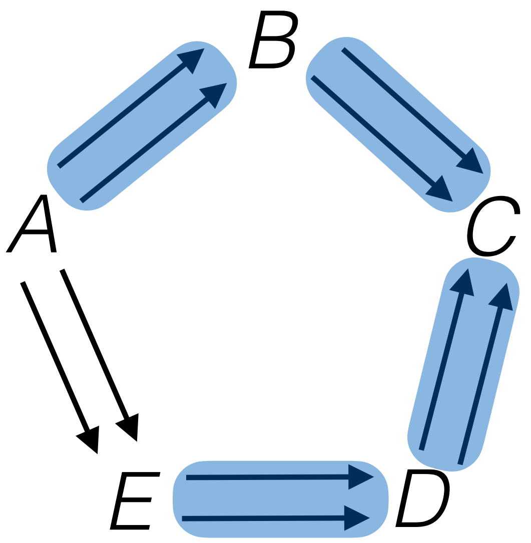
q = 3: # $\leftleftarrows$

p = 2: # $\rightrightarrows$

Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

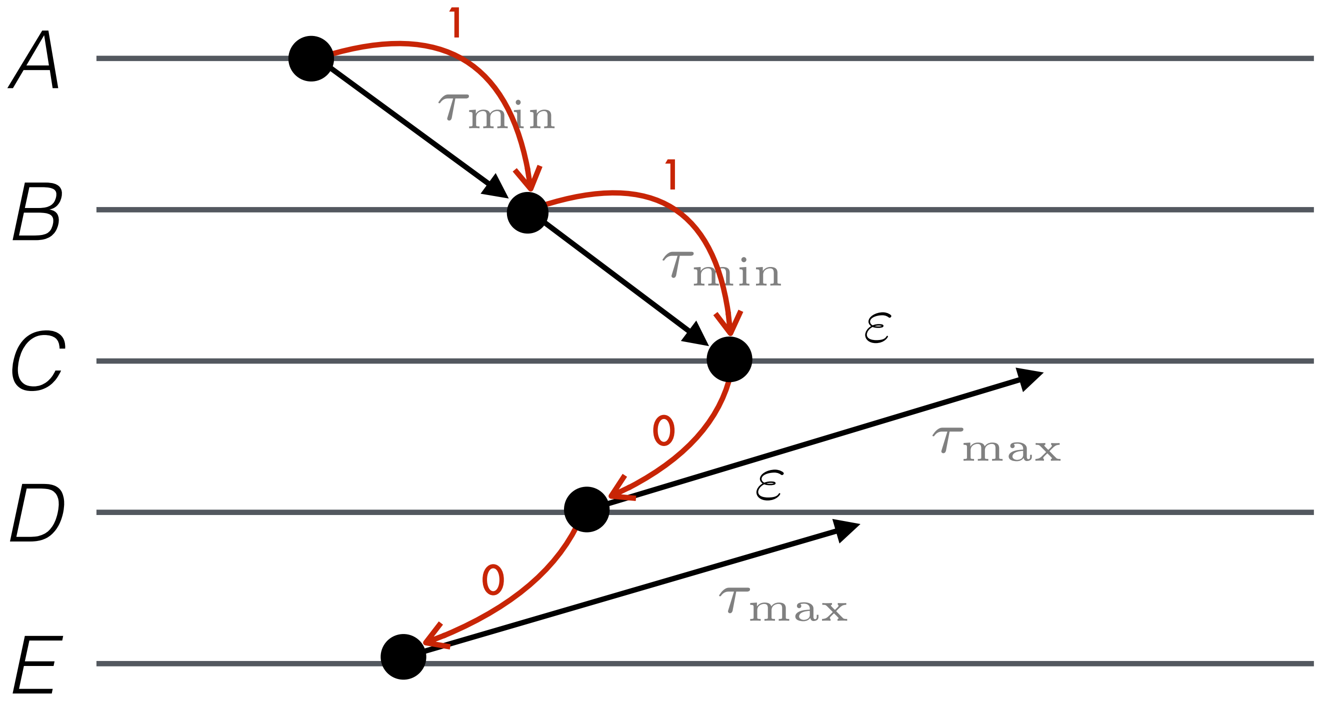
Communications



q = 3: # $\leftleftarrows$

p = 2: # $\rightrightarrows$

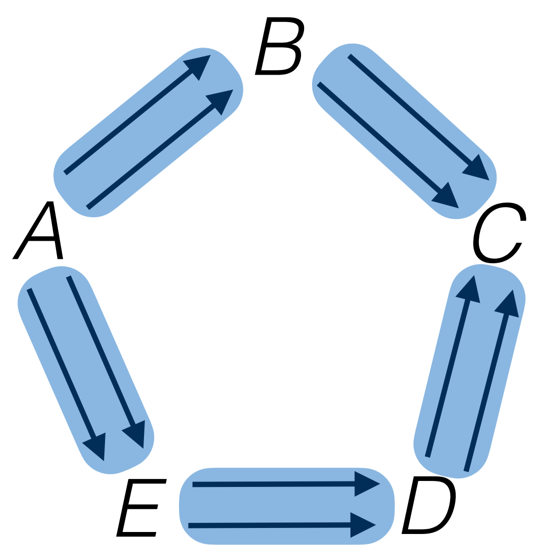
$$q > p \Longrightarrow \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$$



Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

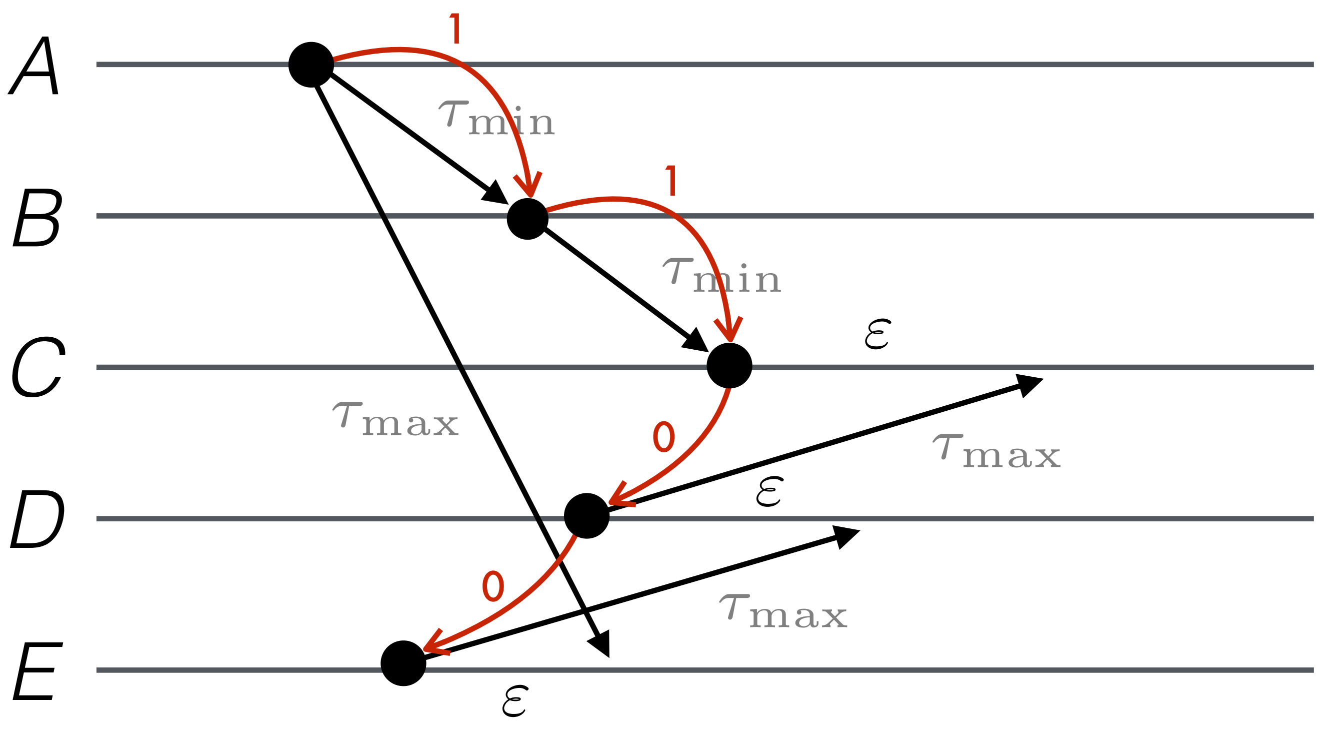
Communications



$q = 3$: # ⇇

$p = 2$: # ⇉

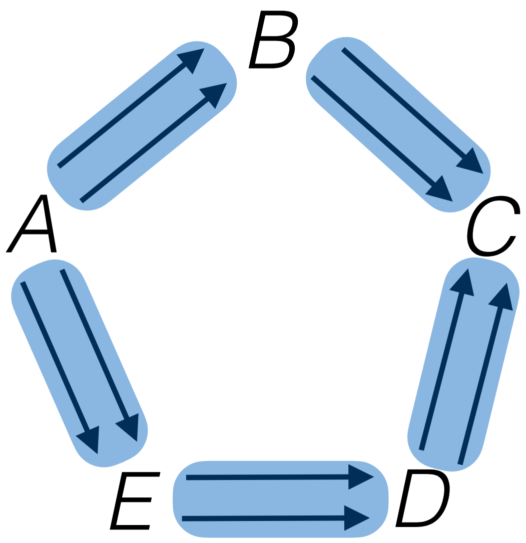
$$q > p \implies \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$$



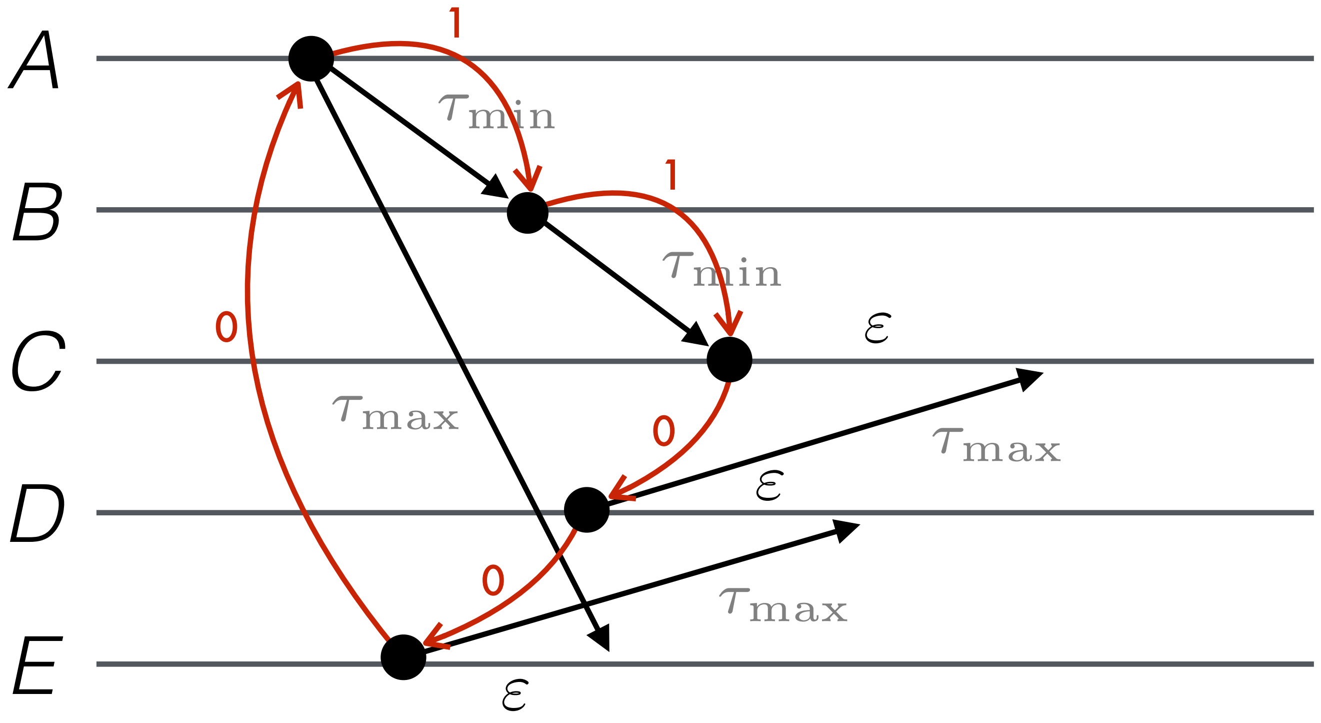
Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

Communications



$$q > p \implies \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$$



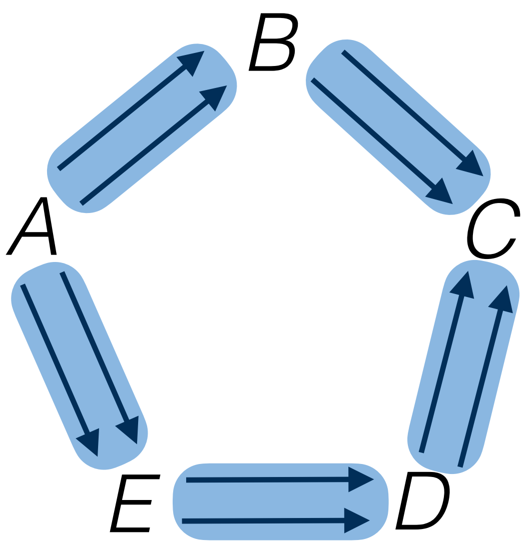
$q = 3$: # ⇇

$p = 2$: # ⇉

Recovering Soundness

Proof: If there is a u-cycle, construction of a counter-example

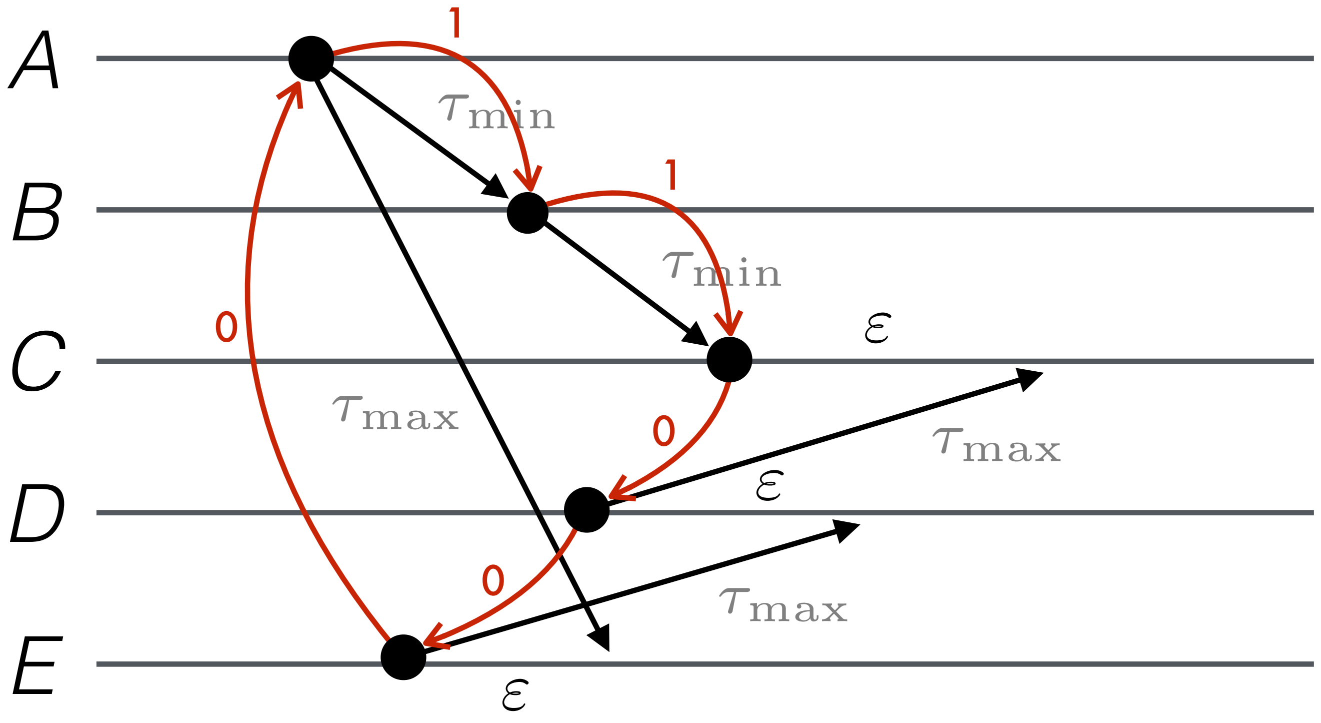
Communications



q = 3: # $\leftleftarrows$

p = 2: # $\rightrightarrows$

$$q > p \implies \varepsilon = (q\tau_{\max} - p\tau_{\min})/q > 0$$



We built a cycle of positive weight!

Recovering Soundness

Proof: On the other hand, by contraposition,

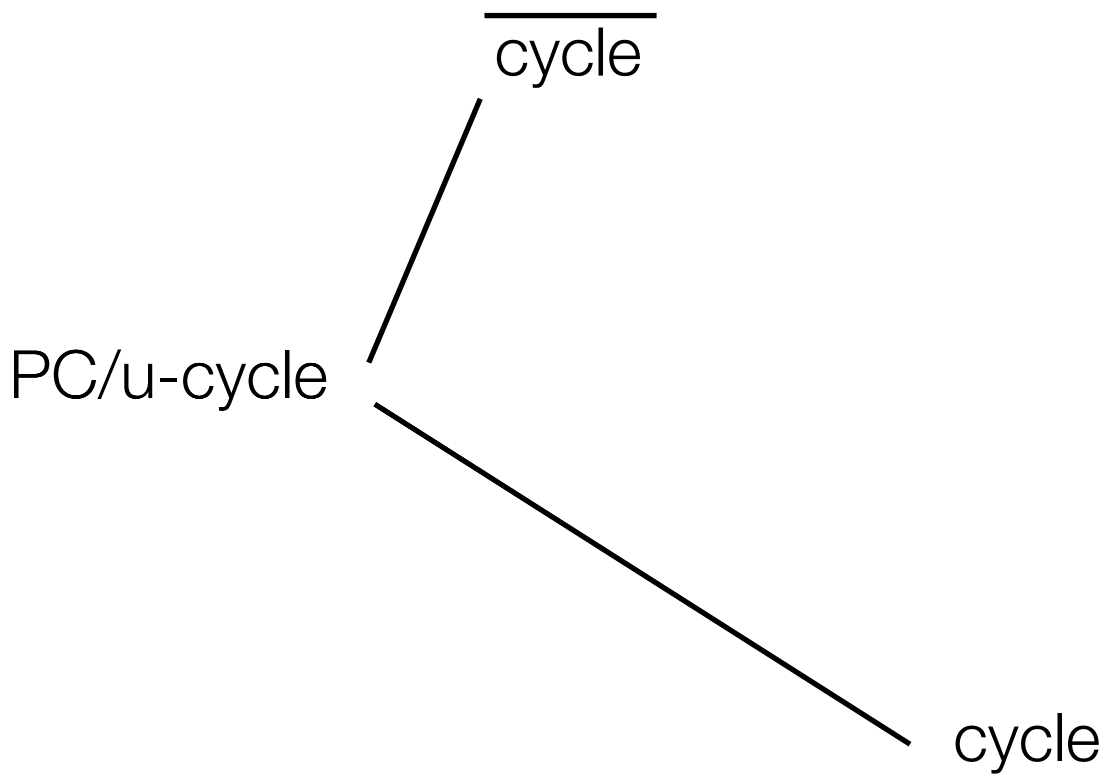
Recovering Soundness

Proof: On the other hand, by contraposition,

PC/u-cycle

Recovering Soundness

Proof: On the other hand, by contraposition,



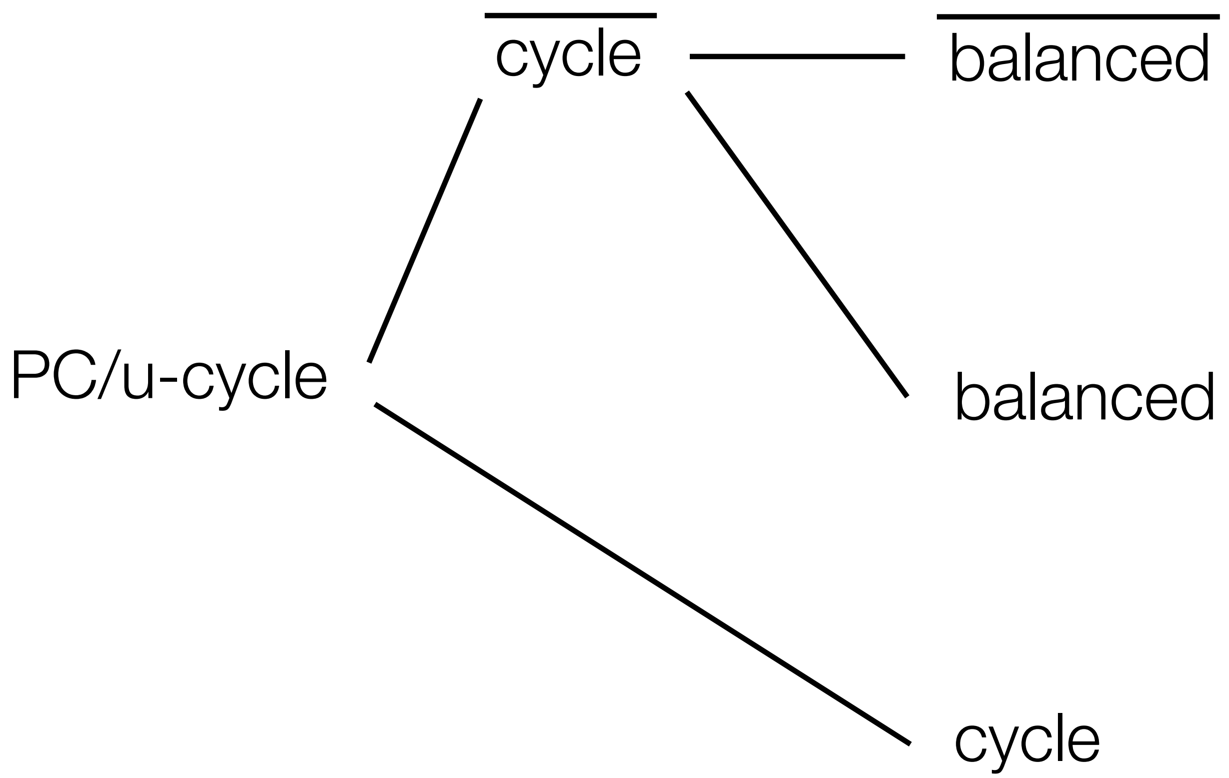
$\overline{\text{cycle}}$

PC/u-cycle

cycle

Recovering Soundness

Proof: On the other hand, by contraposition,



$\overline{\text{cycle}}$

$\overline{\text{balanced}}$

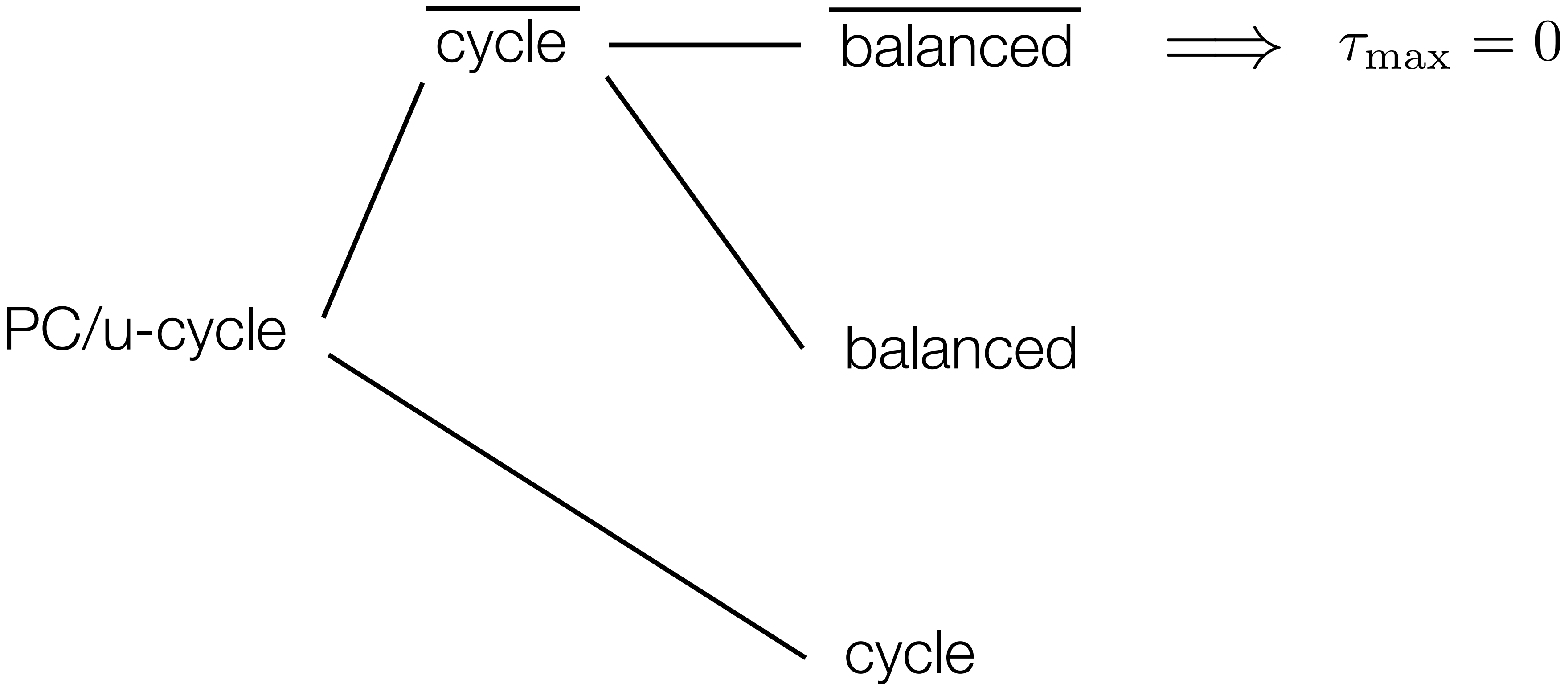
PC/u-cycle

balanced

cycle

Recovering Soundness

Proof: On the other hand, by contraposition,



$\overline{\text{cycle}}$

$\overline{\text{balanced}}$ $\implies \tau_{\max} = 0$

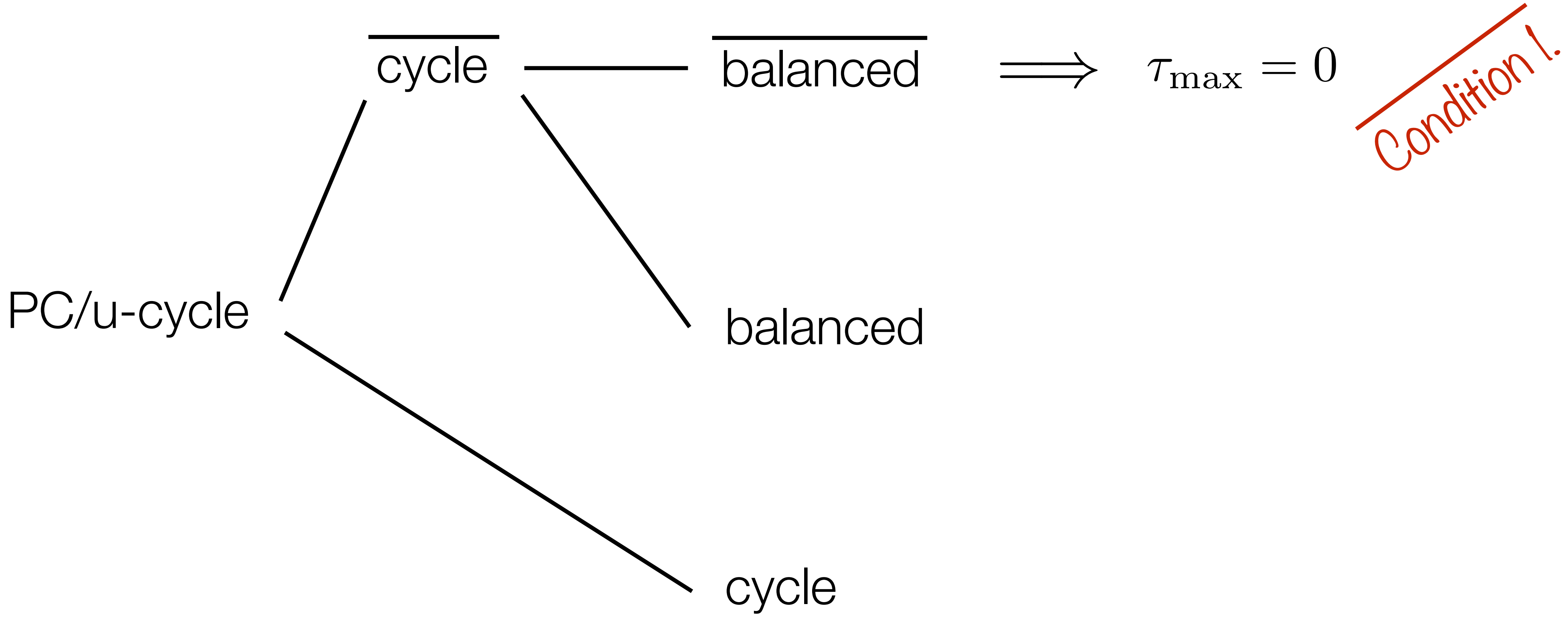
PC/u-cycle

balanced

cycle

Recovering Soundness

Proof: On the other hand, by contraposition,



$\overline{\text{cycle}}$ — $\overline{\text{balanced}}$ $\Longrightarrow$ $\tau_{\max} = 0$

Condition 1.

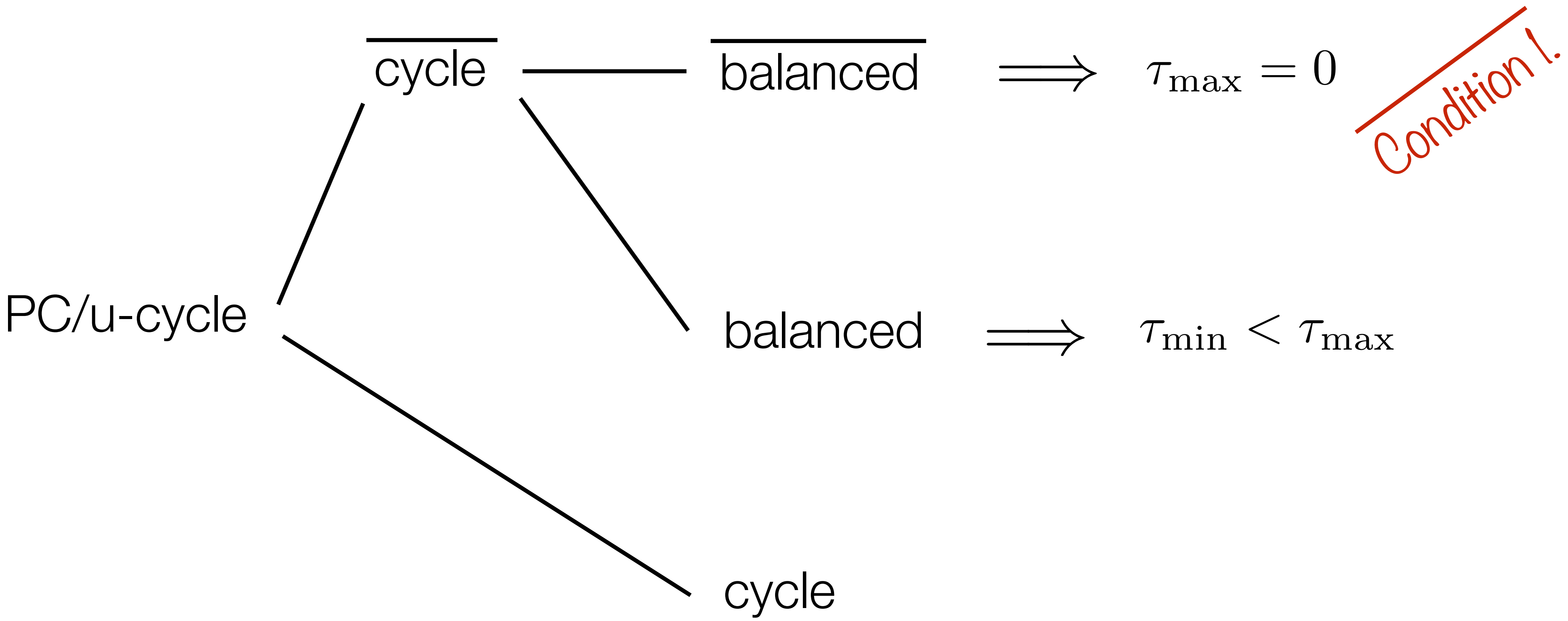
PC/u-cycle

balanced

cycle

Recovering Soundness

Proof: On the other hand, by contraposition,



PC/u-cycle

$\overline{\text{cycle}}$

$\overline{\text{balanced}}$ $\implies$ $\tau_{\max} = 0$

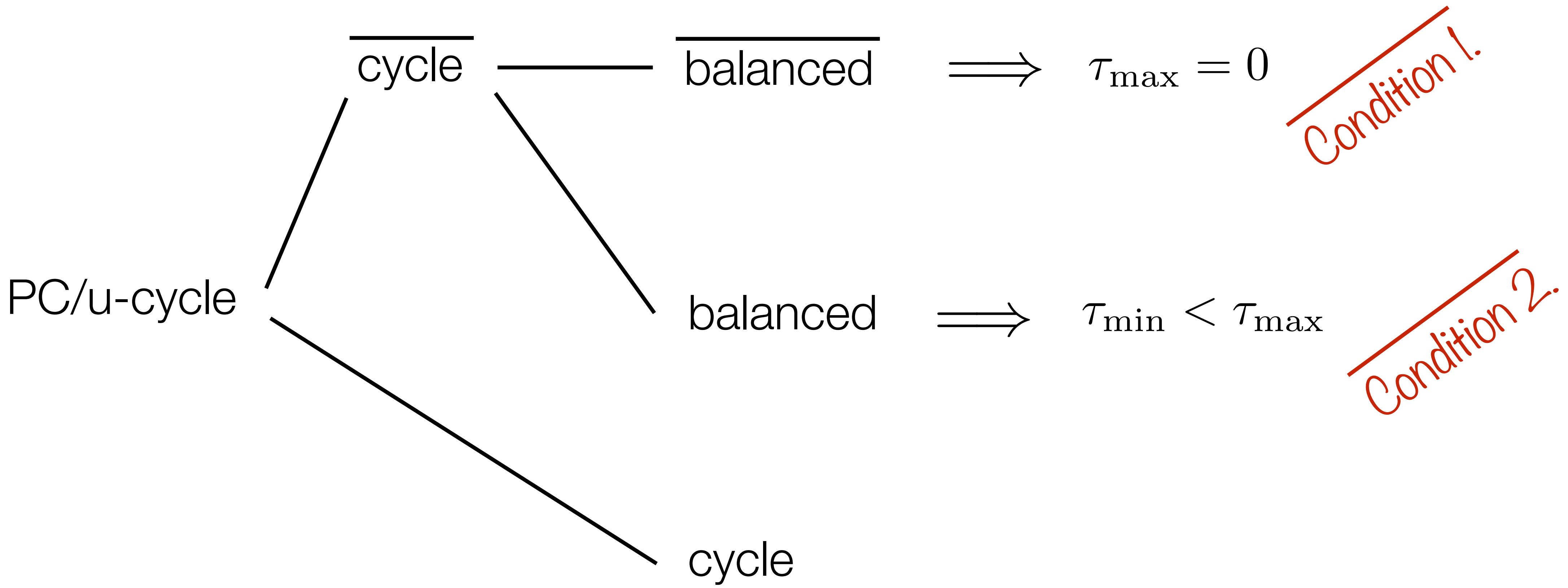
$\overline{\text{Condition 1.}}$

balanced $\implies$ $\tau_{\min} < \tau_{\max}$

cycle

Recovering Soundness

Proof: On the other hand, by contraposition,



PC/u-cycle

$\overline{\text{cycle}}$

$\overline{\text{balanced}}$ $\Longrightarrow$ $\tau_{\max} = 0$

Condition 1.

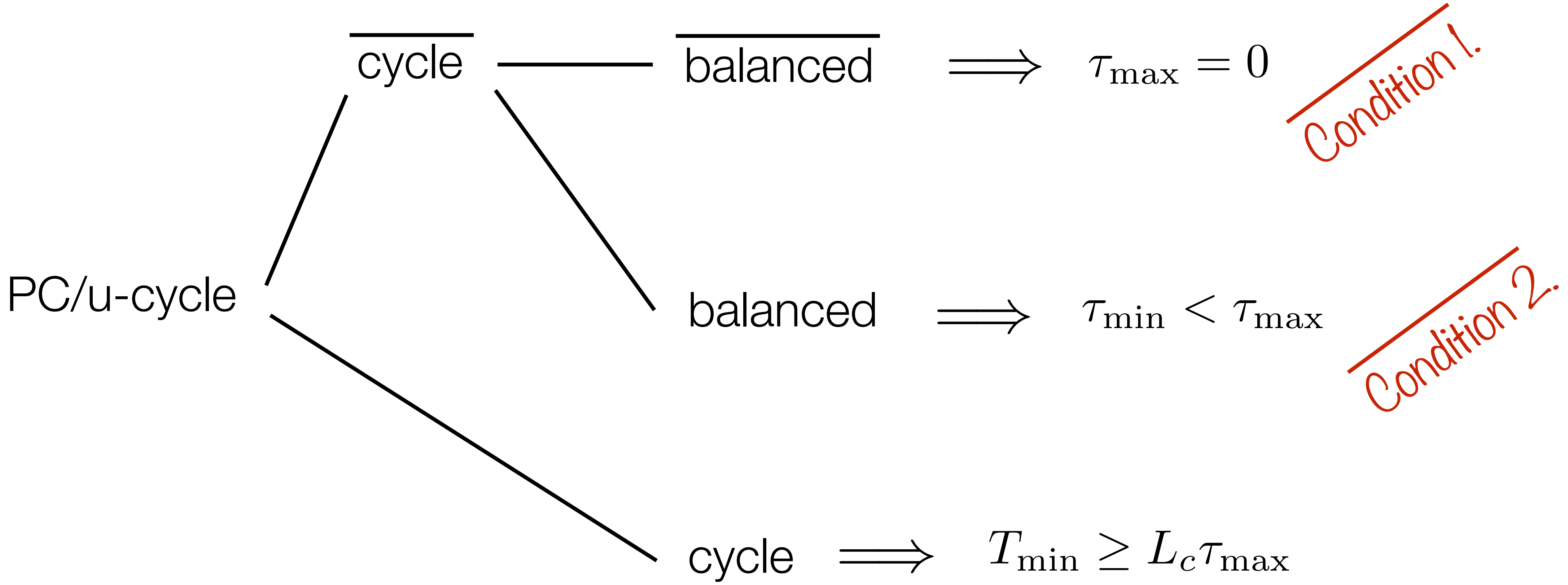
balanced $\Longrightarrow$ $\tau_{\min} < \tau_{\max}$

Condition 2.

cycle

Recovering Soundness

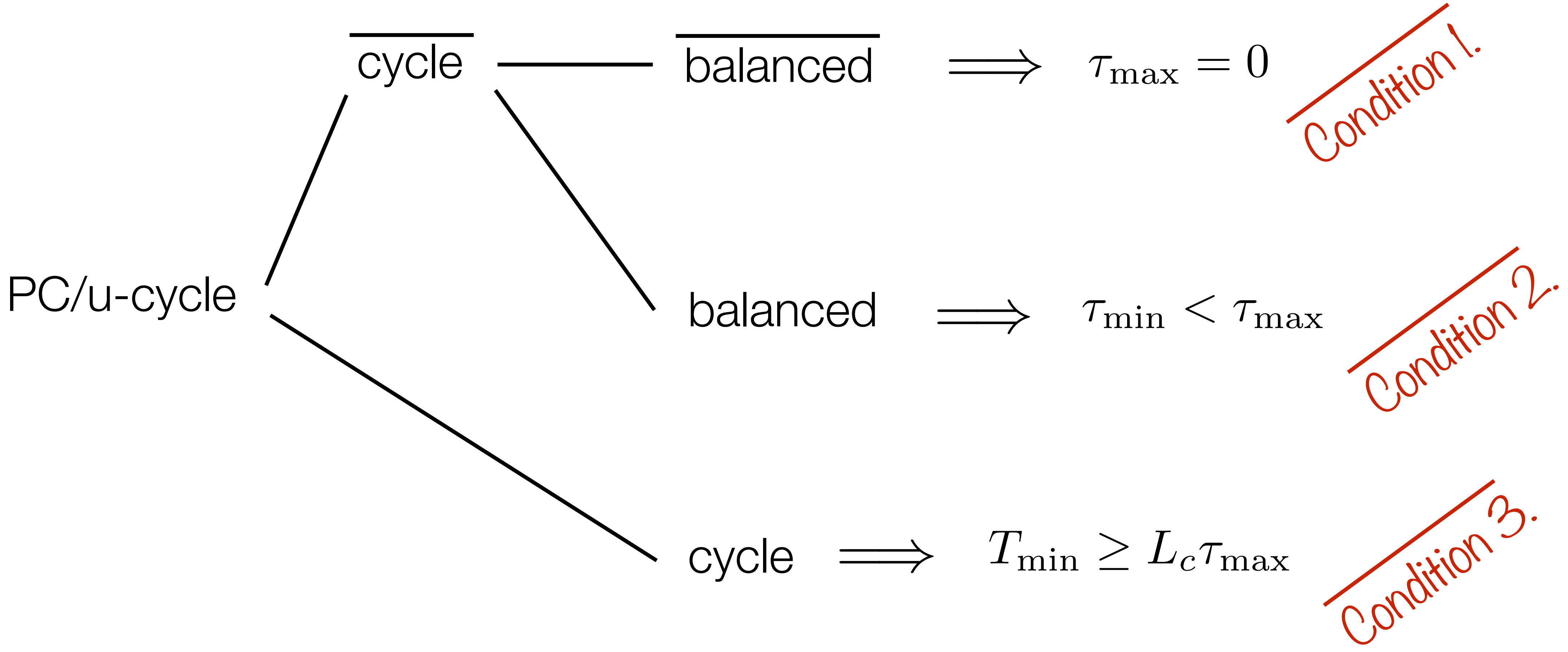
Proof: On the other hand, by contraposition,



PC/u-cycle
$\overline{\text{cycle}}$
$\overline{\text{balanced}} \implies \tau_{\max} = 0$
Condition 1.
balanced $\implies \tau_{\min} < \tau_{\max}$
Condition 2.
cycle $\implies T_{\min} \geq L_c \tau_{\max}$

Recovering Soundness

Proof: On the other hand, by contraposition,



PC/u-cycle

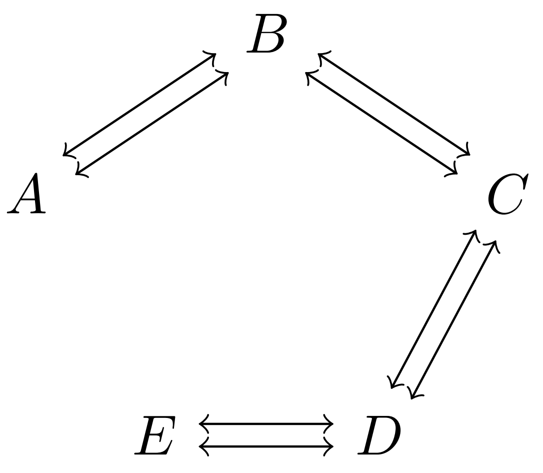
$\overline{\text{cycle}}$ — $\overline{\text{balanced}}$ $\Longrightarrow$ $\tau_{\max} = 0$ Condition 1.

balanced $\Longrightarrow$ $\tau_{\min} < \tau_{\max}$ Condition 2.

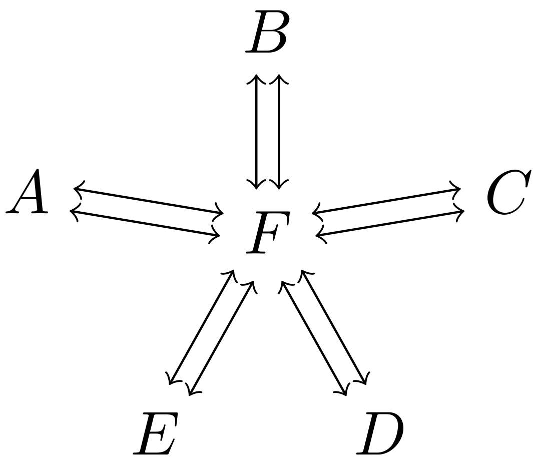
cycle $\Longrightarrow$ $T_{\min} \geq L_c \tau_{\max}$ Condition 3.

Topology Examples

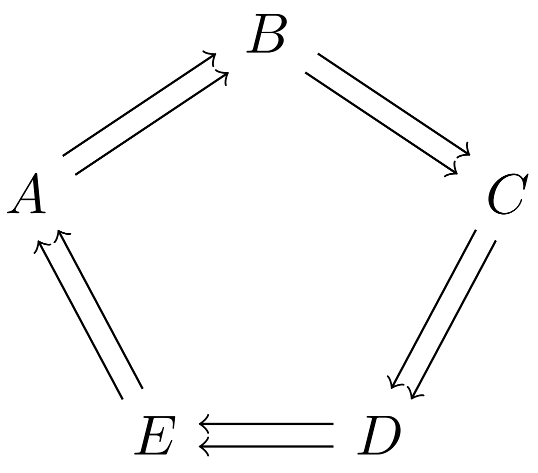
Communications of the application



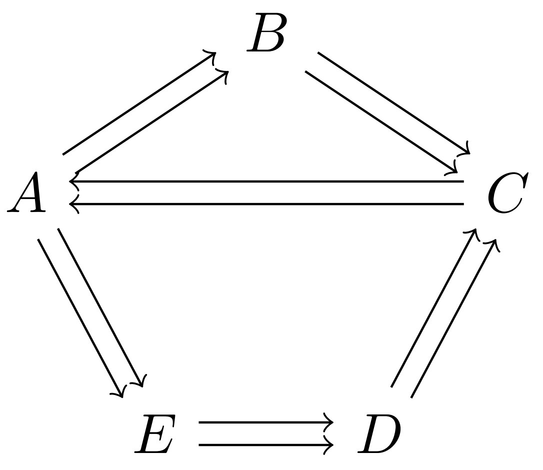
Line : $T_{\min} \geq 2\tau_{\max}$



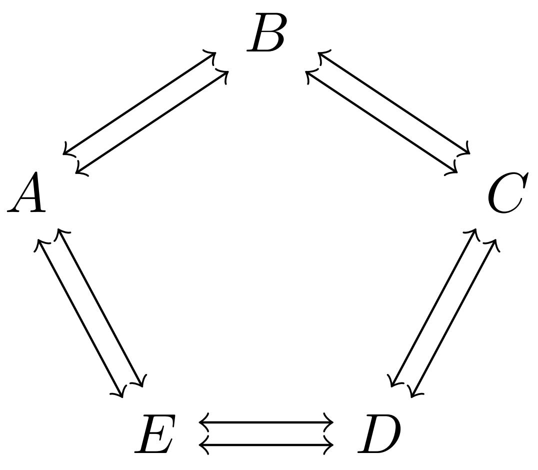
Star : $T_{\min} \geq 2\tau_{\max}$



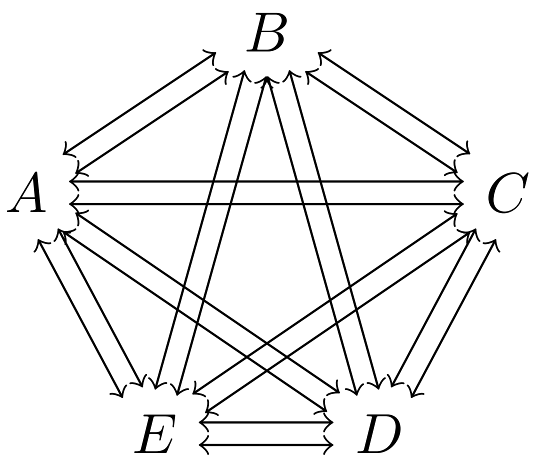
Ring : $T_{\min} \geq 5\tau_{\max}$



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



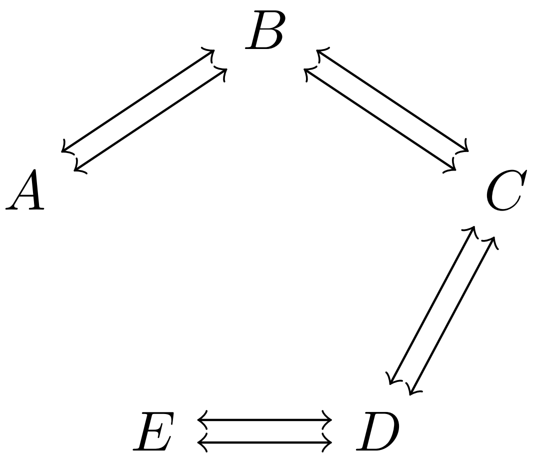
Double ring : $\tau_{\max} = 0$



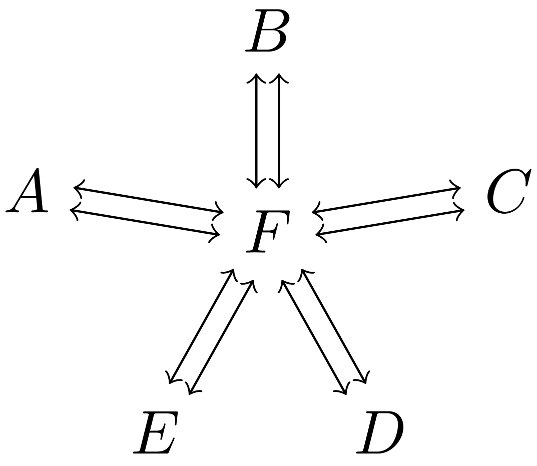
Clique : $\tau_{\max} = 0$

Topology Examples

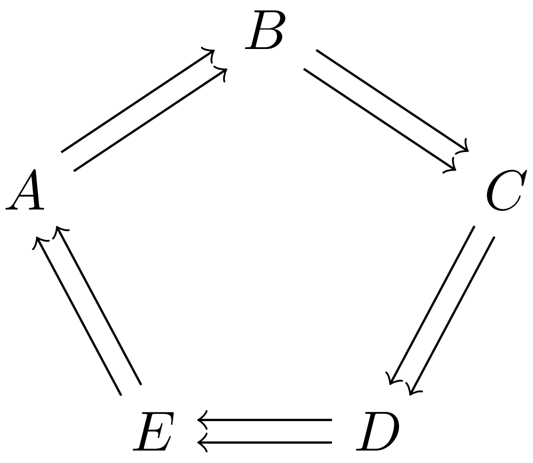
Communications of the application



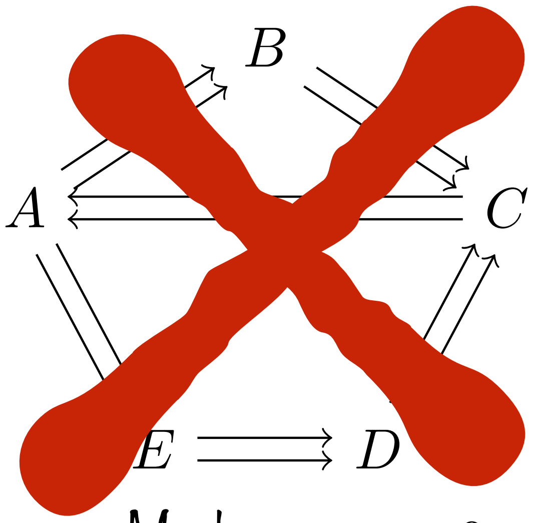
Line : $T_{\min} \geq 2\tau_{\max}$



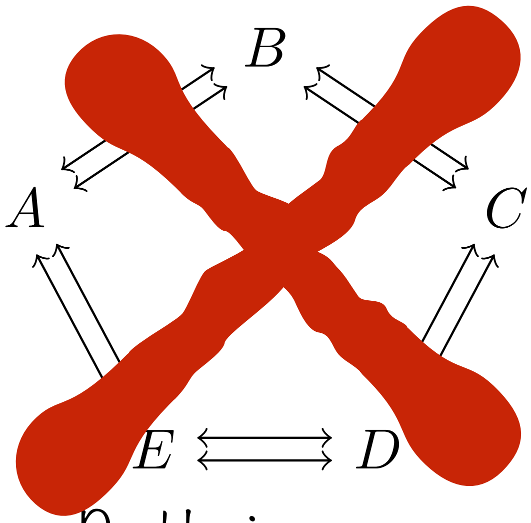
Star : $T_{\min} \geq 2\tau_{\max}$



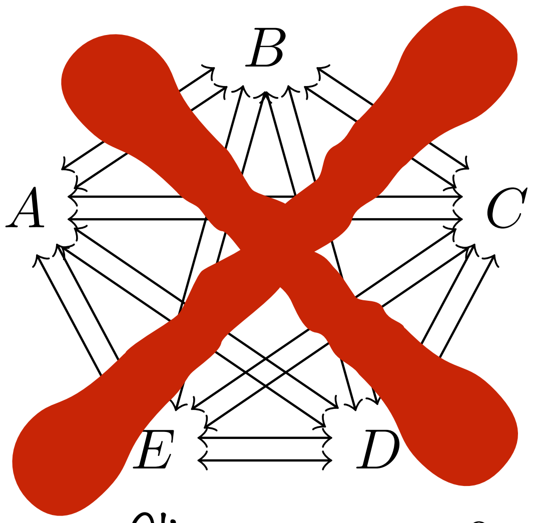
Ring : $T_{\min} \geq 5\tau_{\max}$



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



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



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

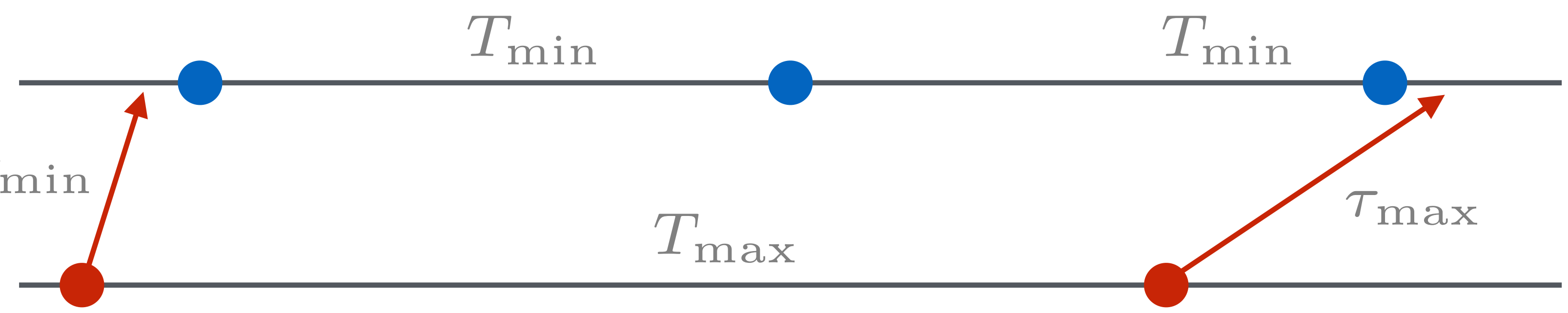
Require instantaneous communications

Quasi-Synchronous Systems

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

Theorem: A real-time model is quasi-synchronous if and only if,

1. it is unitary discretizable
2. $2T_{\text{min}} + \tau_{\text{min}} \geq T_{\text{max}} + \tau_{\text{max}}$



Worst-case scenario

Multirate Systems

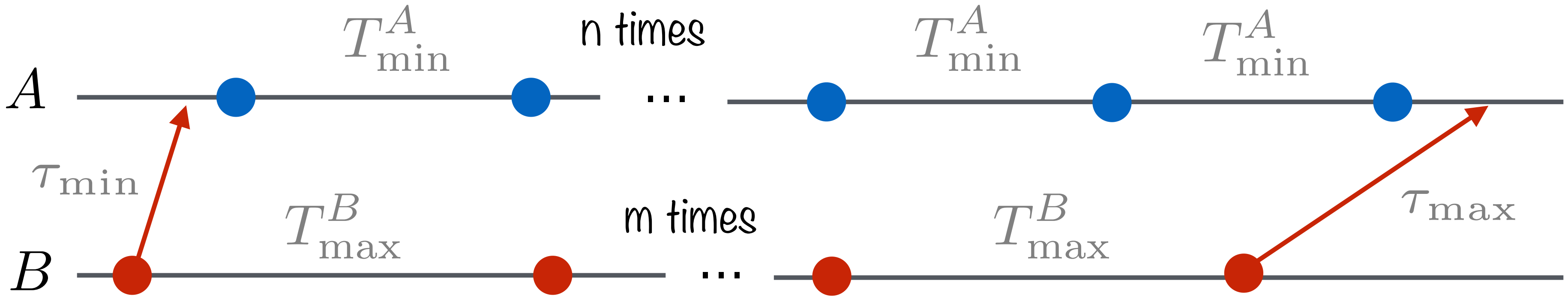
"It is not the case that a component process executes more than **n times between m successive** executions of another process."

n/m-quasi-synchrony [Smeding, Goessler]

Theorem: A real-time model is n/m-quasi-synchronous if and only if,

1. it is unitary discretizable
2. for any pair of communicating nodes $A \leftleftarrows B$

$$nT^A_{\min} + \tau_{\min} \geq (m-1)T^B_{\max} + \tau_{\max}$$
$$nT^B_{\min} + \tau_{\min} \geq (m-1)T^A_{\max} + \tau_{\max}$$



Worst-case scenario

[SG12]

Summary

The quasi-synchronous abstraction:
1. Model transmission as unit delays
2. Constrain node activations interleavings

Contributions:
- Condition 1 is not sound in general
- Notion of unitary discretization
- Exact conditions to recover soundness
- Characterization of quasi-synchronous systems
- Generalization to multirate systems

Soundness of the Quasi-Synchronous Abstraction

Guillaume Baudart*† Timothy Bourke‡* Marc Pouzet*§

* École normale supérieure, PSL Research University
† Sorbonne Universités, UPMC Univ. Paris 06
‡ Inria Paris

[illegible]

Constrain both the communication graph and the real-time characteristics of the architecture to recover soundness of the quasi-synchronous abstraction.

[FMCAD 2016]

Overview

Implementation

Deploying code on
quasi-periodic architectures

Loosely Time-Triggered Architectures

Overview

How to preserve the semantics
of the embedded application?

Implementation

Deploying code on
quasi-periodic architectures

Loosely Time-Triggered Architectures

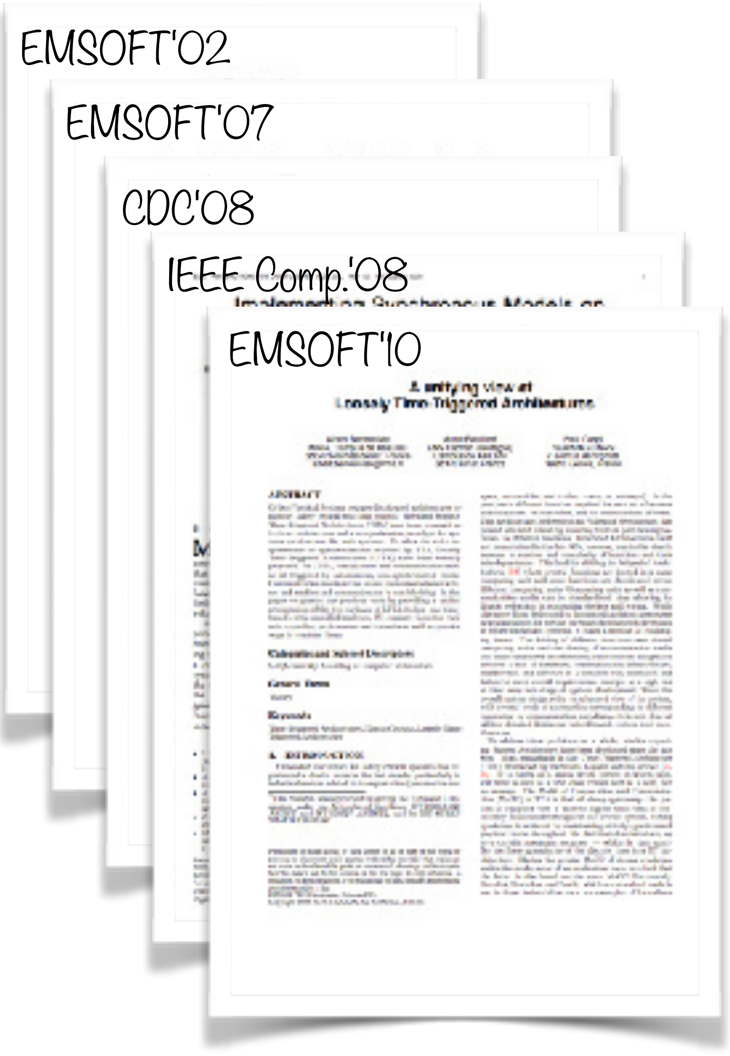
Overview

How to preserve the semantics of the embedded application?

Implementation

Deploying code on
quasi-periodic architectures

Loosely Time-Triggered Architectures



A unifying view of Loosely Time-Triggered Architectures

[Benveniste, Bouillard, Caspi, Di Natale, Pinello, Talpin, Tripakis, Sangiovanni-Vincentelli]

[BCLG+02, BCDN+07, CB08, TPB+08, BBC10]

Overview

How to preserve the semantics of the embedded application?

Implementation

Deploying code on
quasi-periodic architectures

Loosely Time-Triggered Architectures

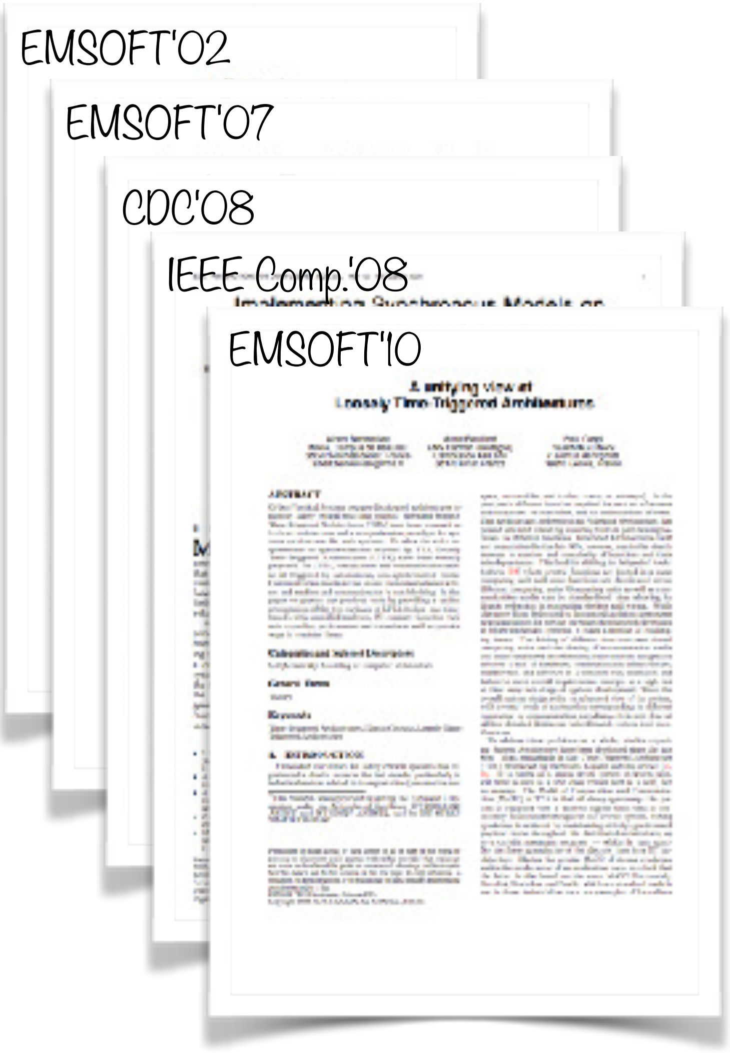
Contributions

Unified synchronous framework

Executable specifications

Correctness proofs

Optimizations and comparisons



A unifying view of
Loosely Time-Triggered Architectures

[Benveniste, Bouillard, Caspi, Di Natale, Pinello, Talpin, Tripakis, Sangiovanni-Vincentelli]

[BCLG+02, BCDN+07, CB08, TPB+08, BBC10]

How to Preserve the Semantics?

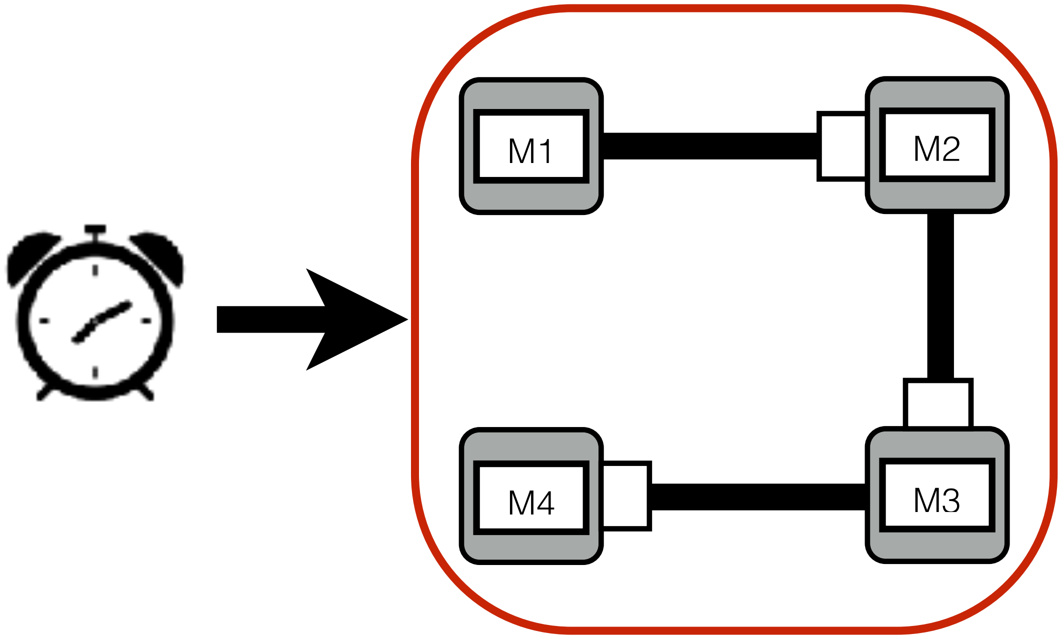
(of an application on a quasi-periodic architecture)

How to Preserve the Semantics?

(of an application on a quasi-periodic architecture)

Clock synchronization

e.g. TTA [Kopetz, Bauer 2003]



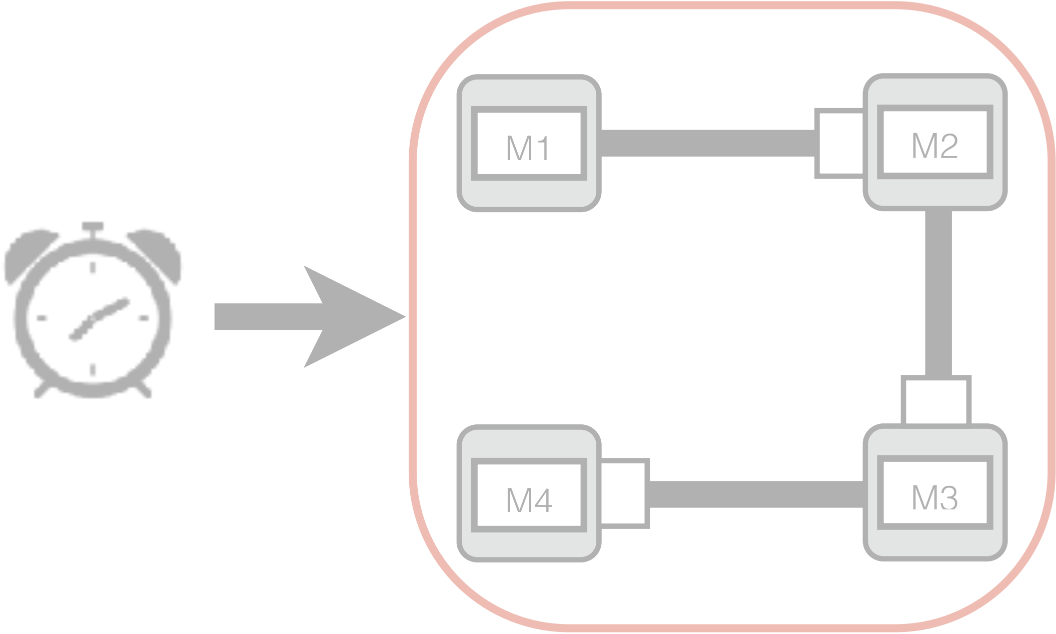
Require dedicated hardware and dedicated controllers

How to Preserve the Semantics?

(of an application on a quasi-periodic architecture)

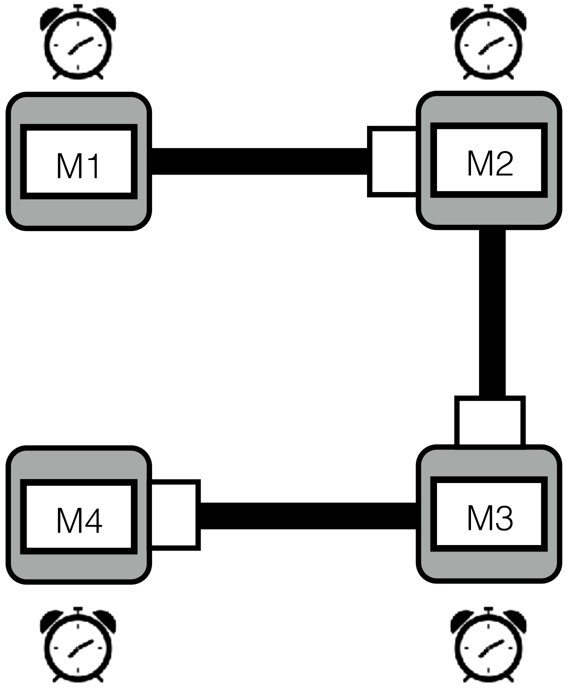
Clock synchronization

e.g. TTA [Kopetz, Bauer 2003]



Require dedicated hardware and dedicated controllers

Unsynchronized nodes + Middleware = LTTA

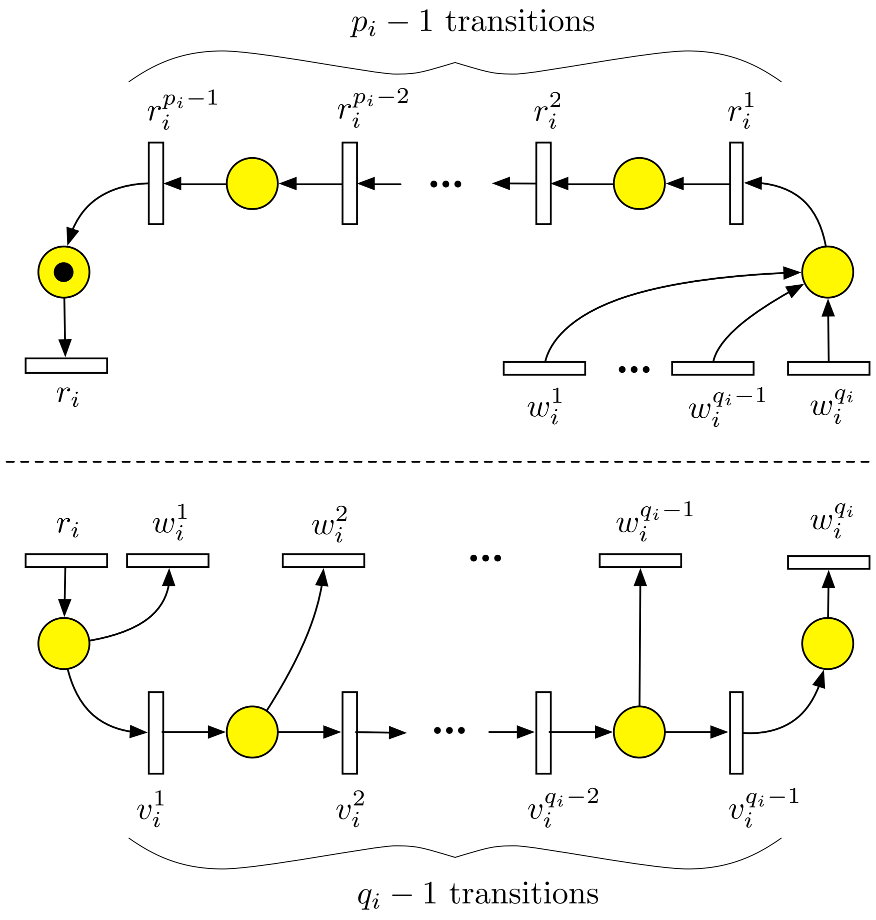


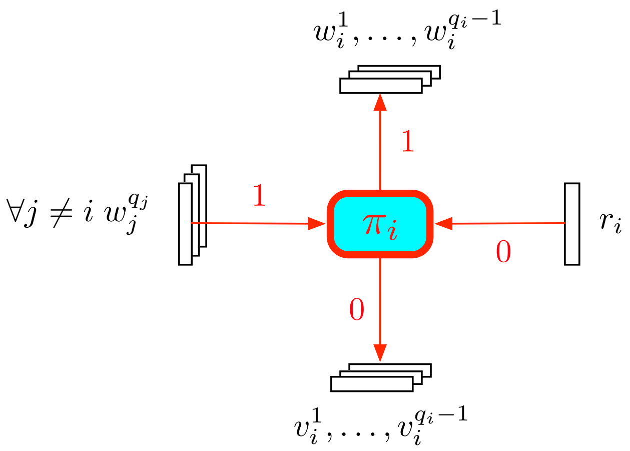
Lightweight alternative

A Synchronous Framework

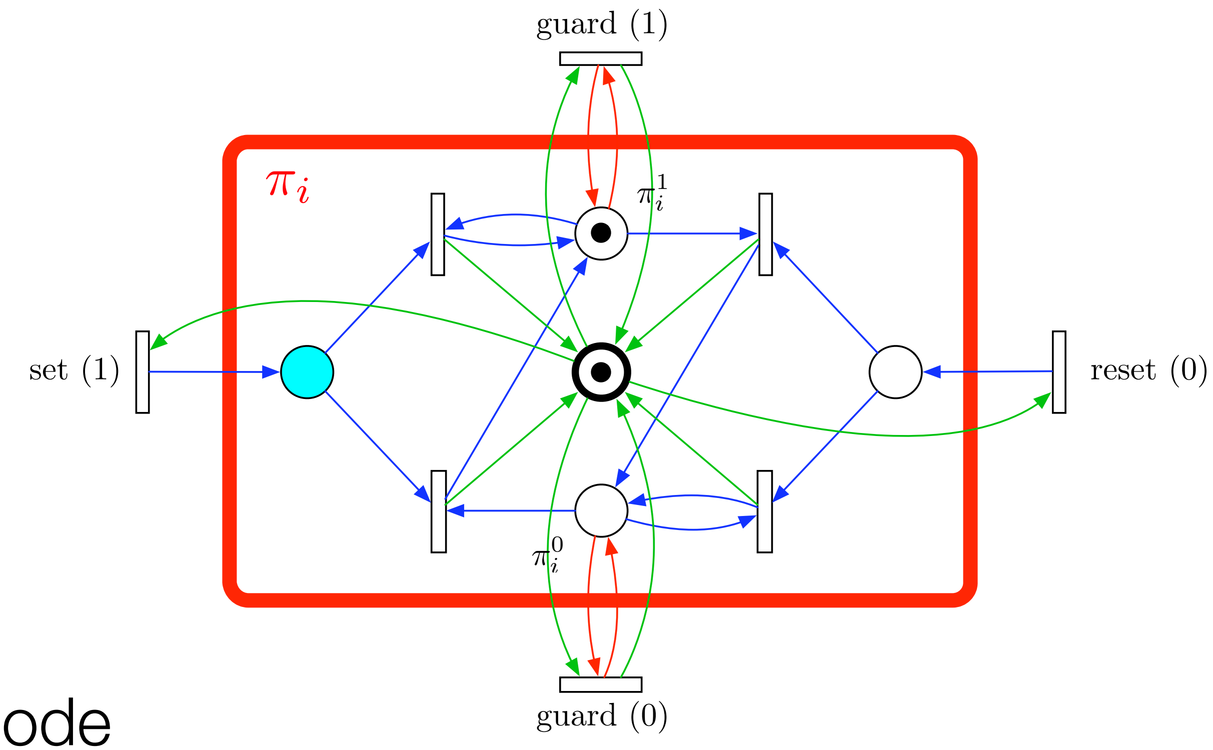
Previous model: timed Petri nets

[Benveniste, Caspi, Bouillard]





$w_i^1, \ldots, w_i^{q_i-1}$
1
$\forall j \neq i \; w_j^{q_j}$
1
π_i
0
r_i
0
$v_i^1, \ldots, v_i^{q_i-1}$



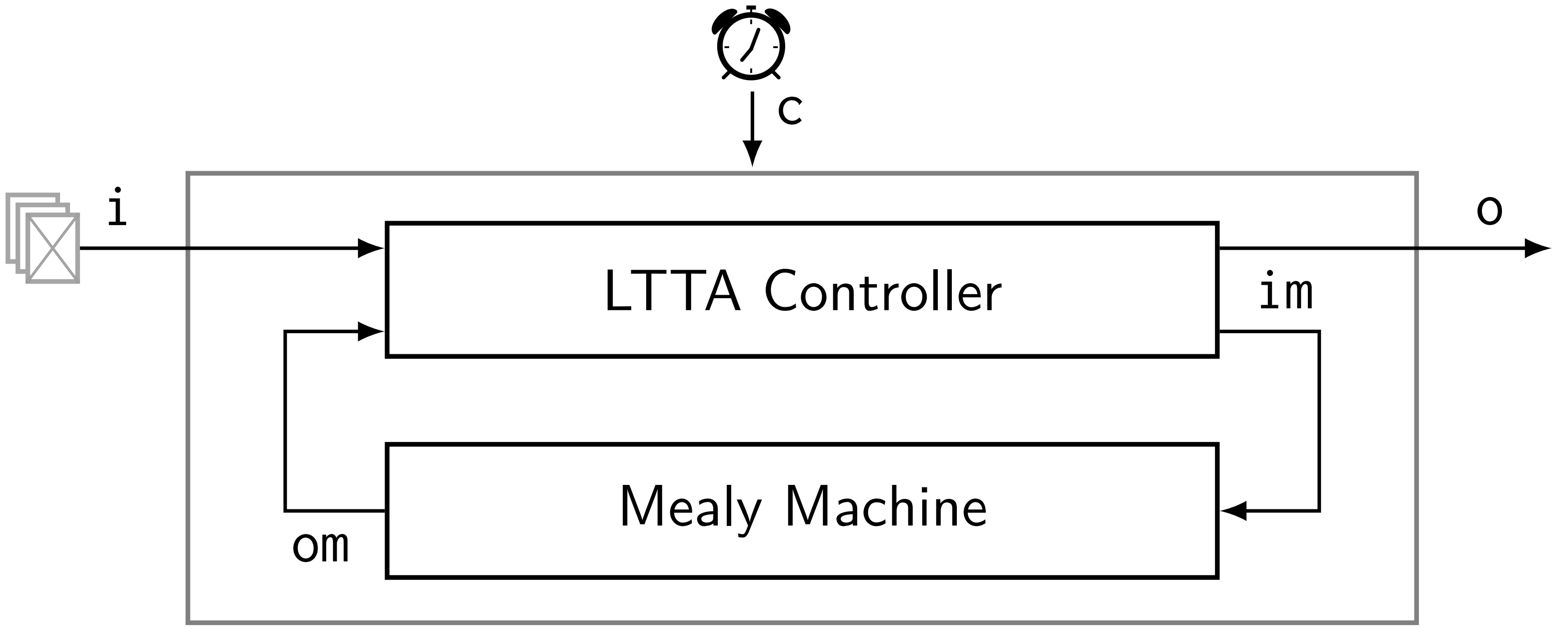
Help: Design the protocol
Analysis (worst case throughput)

But: Cannot be compiled/simulated
Mix real-time characteristics and discrete code

[BBC10, BBBC14]

A Synchronous Framework

A **middleware** controls the execution of the embedded application
The controller **waits** for new inputs and **delays** publications

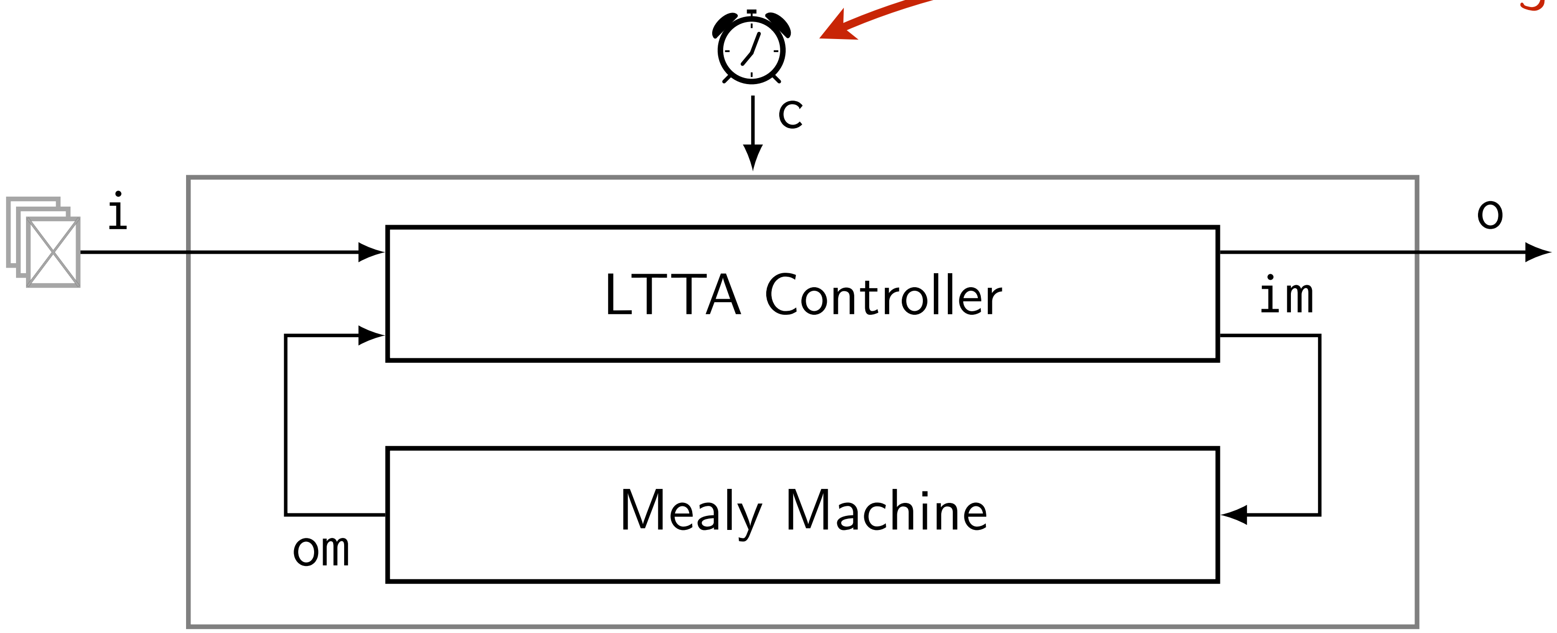


```
let node ltta_node(i) = o where
  rec (o, im) = ltta_controller(i, om)
  and present im(v) → do emit om = machine(v) done
```

Shell wrapper: Latency insensitive design (LID)
[Carloni, McMillan, Sangiovanni-Vincentelli]

A Synchronous Framework

A **middleware** controls the execution of the embedded application
The controller **waits** for new inputs and **delays** publications



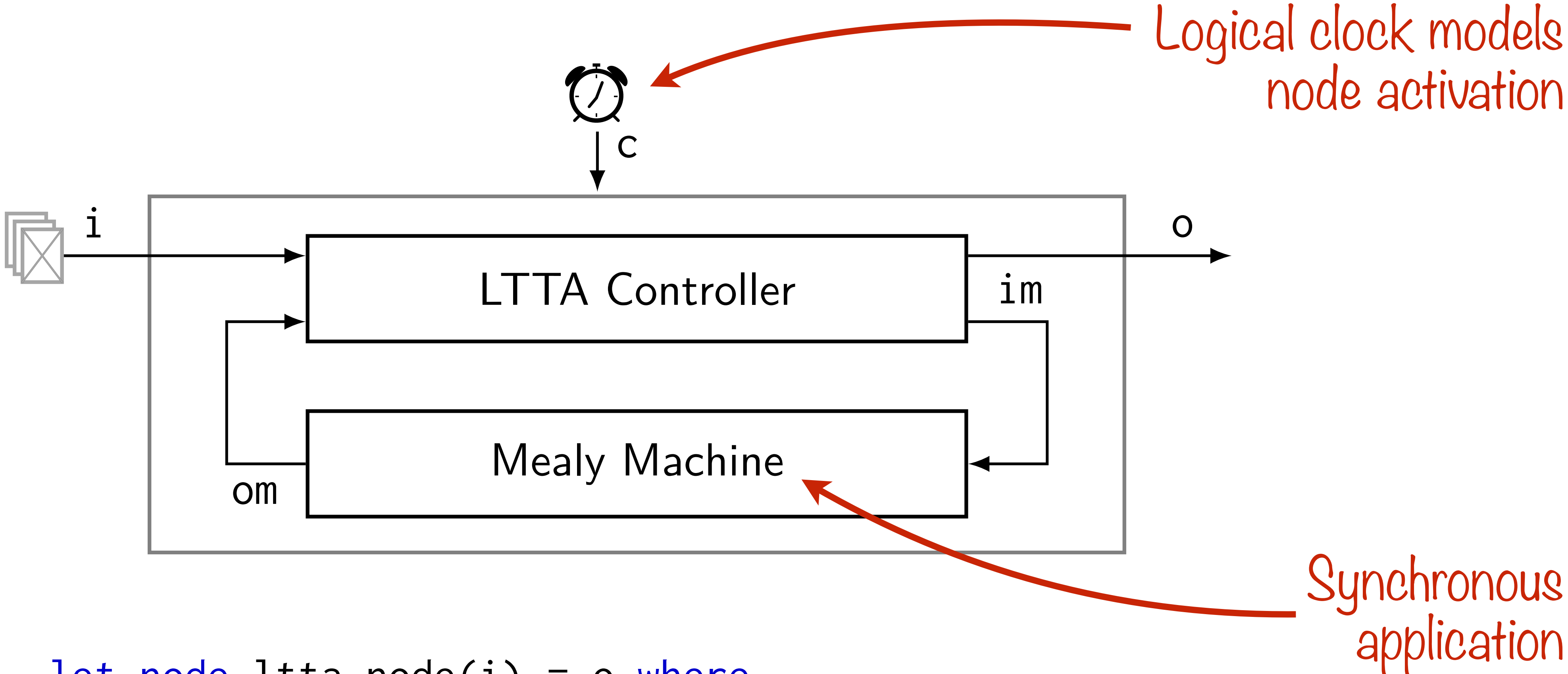
Logical clock models
node activation

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

Shell wrapper: Latency insensitive design (LID)
[Carloni, McMillan, Sangiovanni-Vincentelli]

A Synchronous Framework

A **middleware** controls the execution of the embedded application
The controller **waits** for new inputs and **delays** publications



Logical clock models node activation

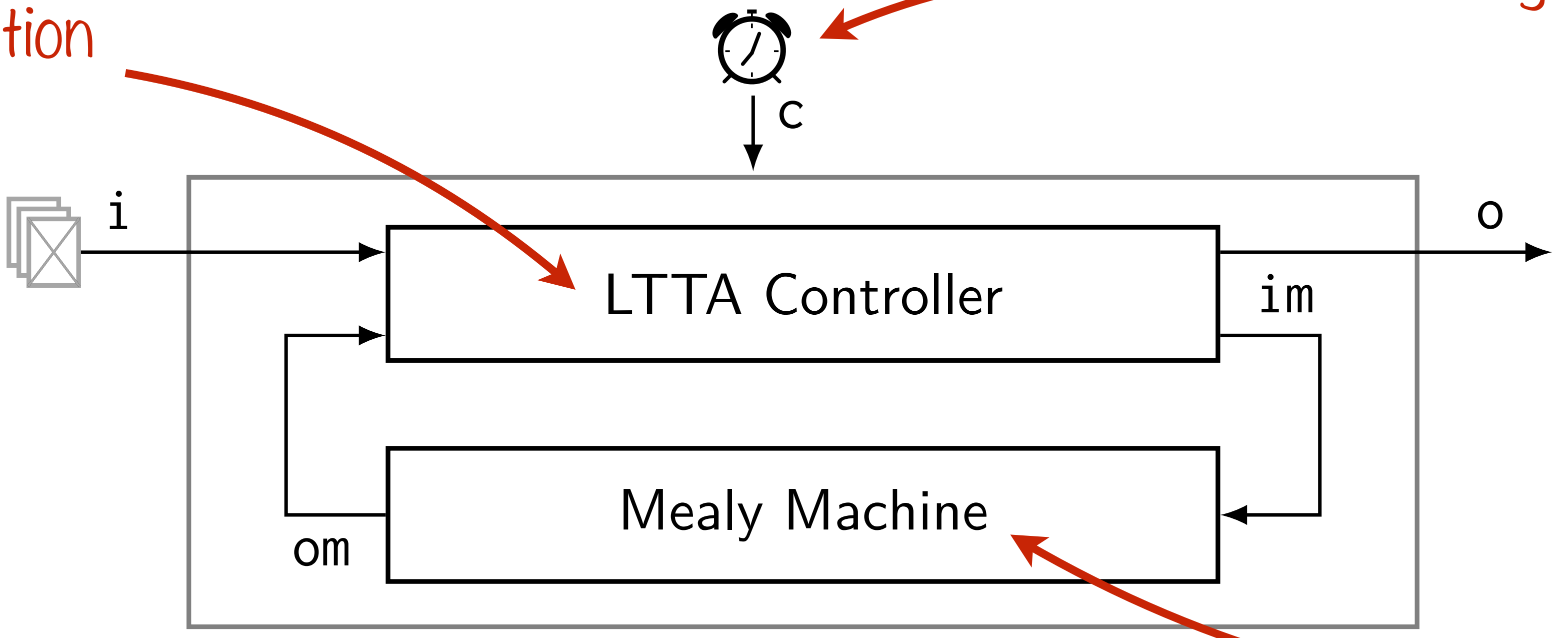
Synchronous application

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

Shell wrapper: Latency insensitive design (LID)
[Carloni, McMillan, Sangiovanni-Vincentelli]

A Synchronous Framework

A **middleware** controls the execution of the embedded application
The controller **waits** for new inputs and **delays** publications



Controls the execution of the application

Logical clock models node activation

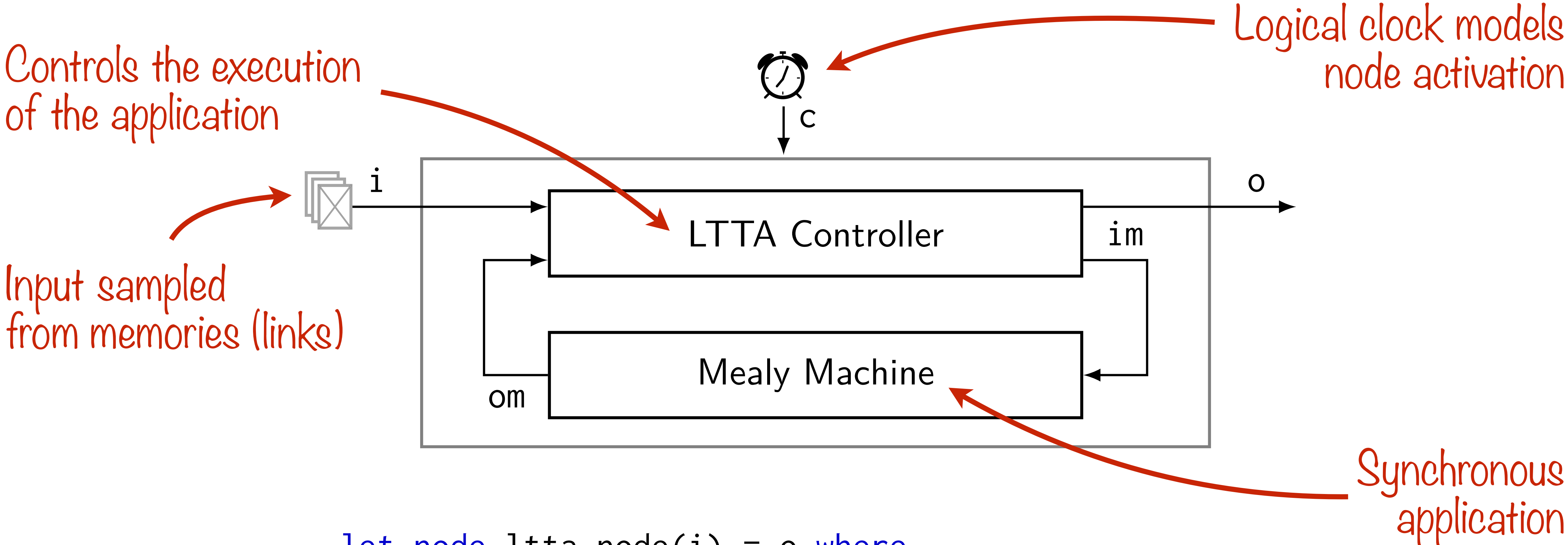
Synchronous application

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

Shell wrapper: Latency insensitive design (LID)
[Carloni, McMillan, Sangiovanni-Vincentelli]

A Synchronous Framework

A **middleware** controls the execution of the embedded application
The controller **waits** for new inputs and **delays** publications



Controls the execution of the application

Logical clock models node activation

Input sampled from memories (links)

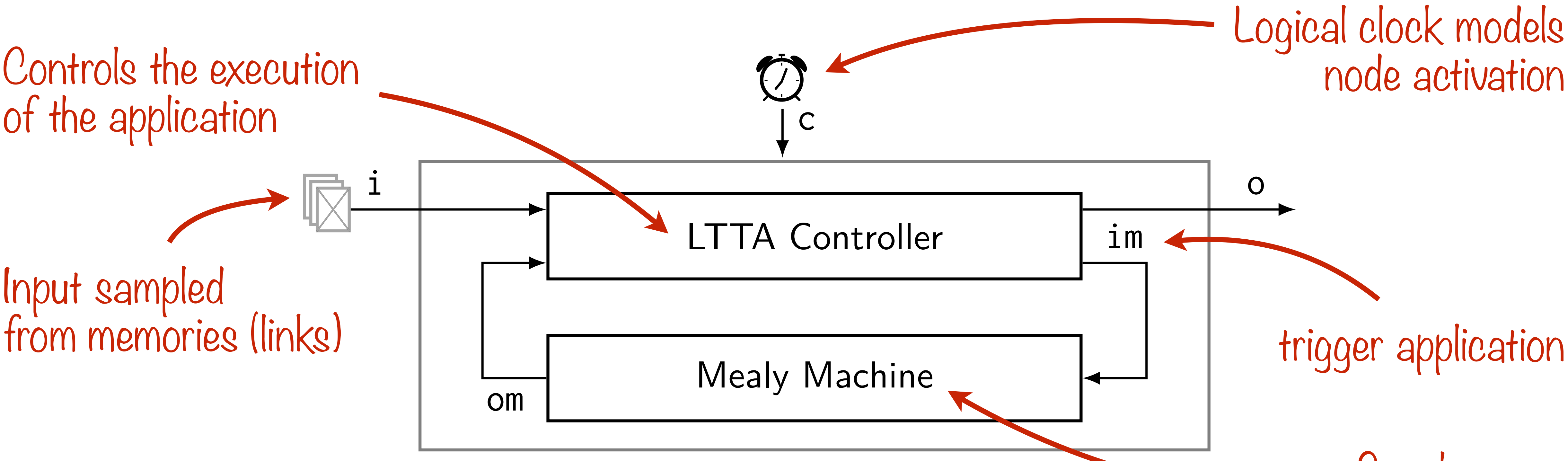
Synchronous application

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

Shell wrapper: Latency insensitive design (LID)
[Carloni, McMillan, Sangiovanni-Vincentelli]

A Synchronous Framework

A **middleware** controls the execution of the embedded application
The controller **waits** for new inputs and **delays** publications

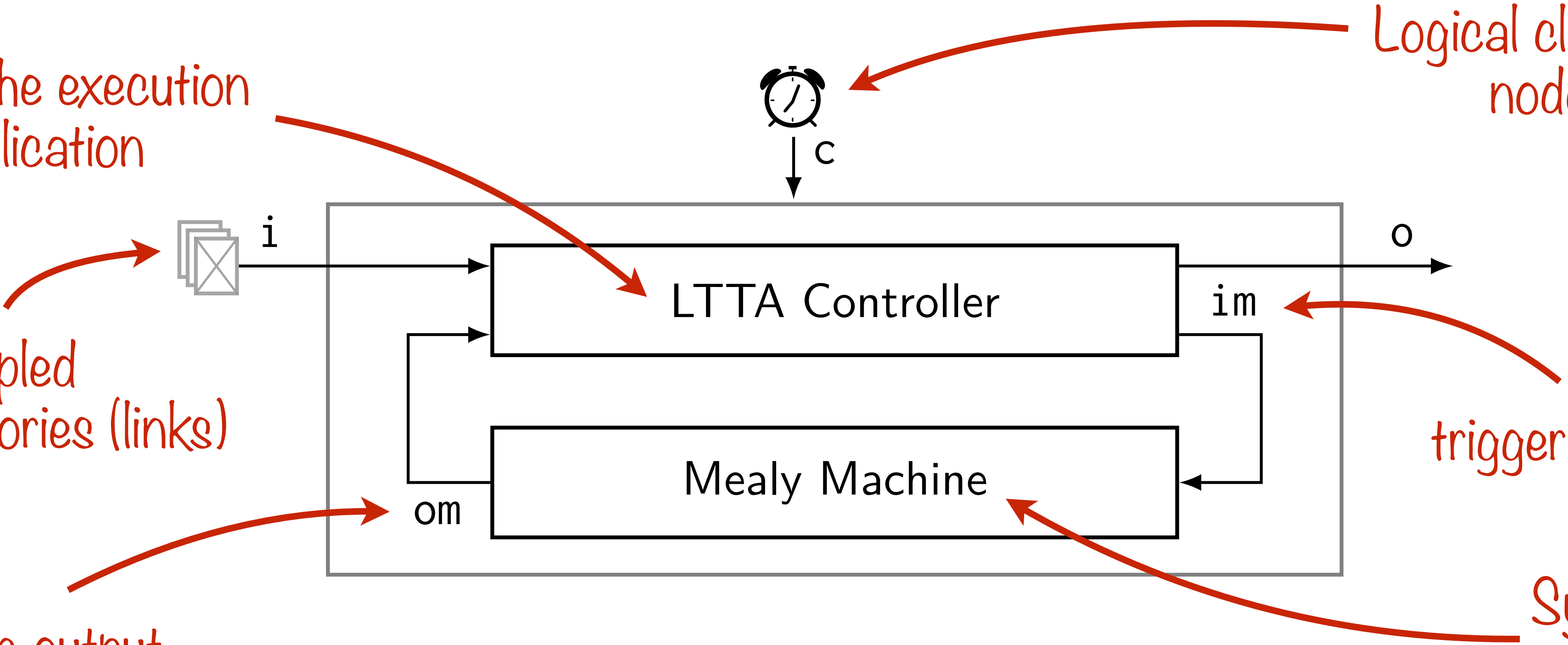

Controls the execution of the application
Logical clock models node activation
c
i
o
LTTA Controller
im
Input sampled from memories (links)
trigger application
Mealy Machine
om
Synchronous application

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

Shell wrapper: Latency insensitive design (LID)
[Carloni, McMillan, Sangiovanni-Vincentelli]

A Synchronous Framework

A **middleware** controls the execution of the embedded application
The controller **waits** for new inputs and **delays** publications



Controls the execution of the application

Logical clock models node activation

Input sampled from memories (links)

trigger application

Application output

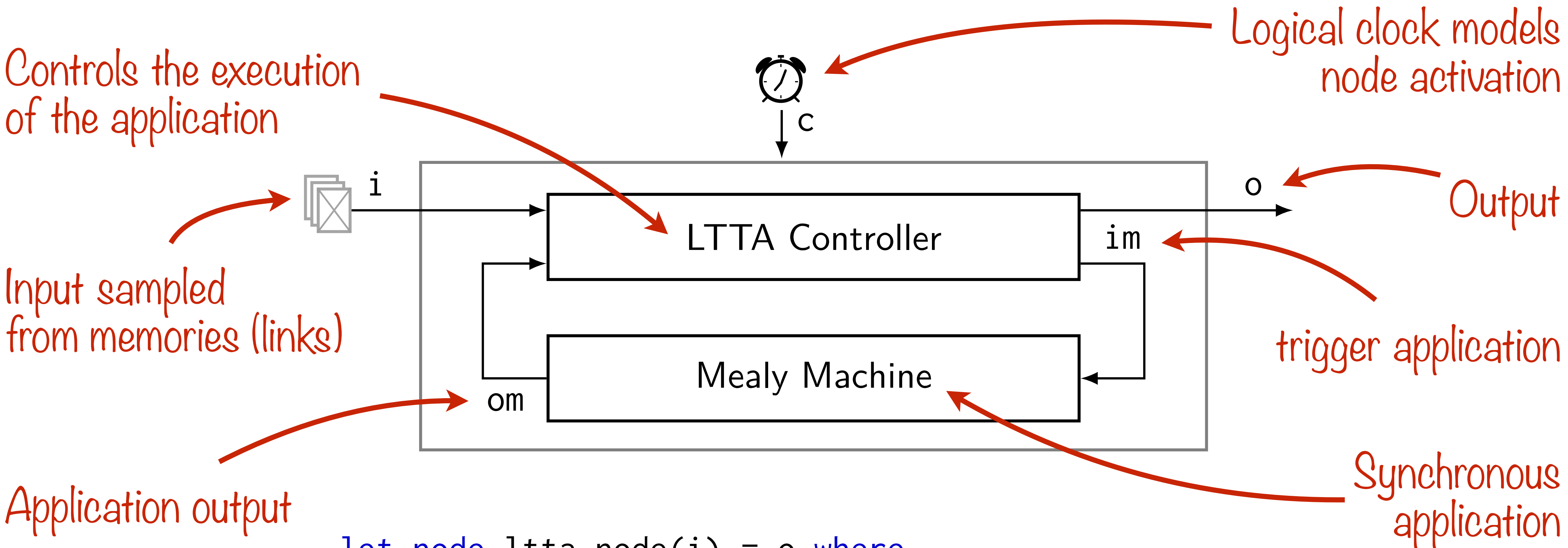
Synchronous application

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

Shell wrapper: Latency insensitive design (LID)
[Carloni, McMillan, Sangiovanni-Vincentelli]

A Synchronous Framework

A **middleware** controls the execution of the embedded application
The controller **waits** for new inputs and **delays** publications



Controls the execution of the application

Logical clock models node activation

Input sampled from memories (links)

Output

trigger application

Application output

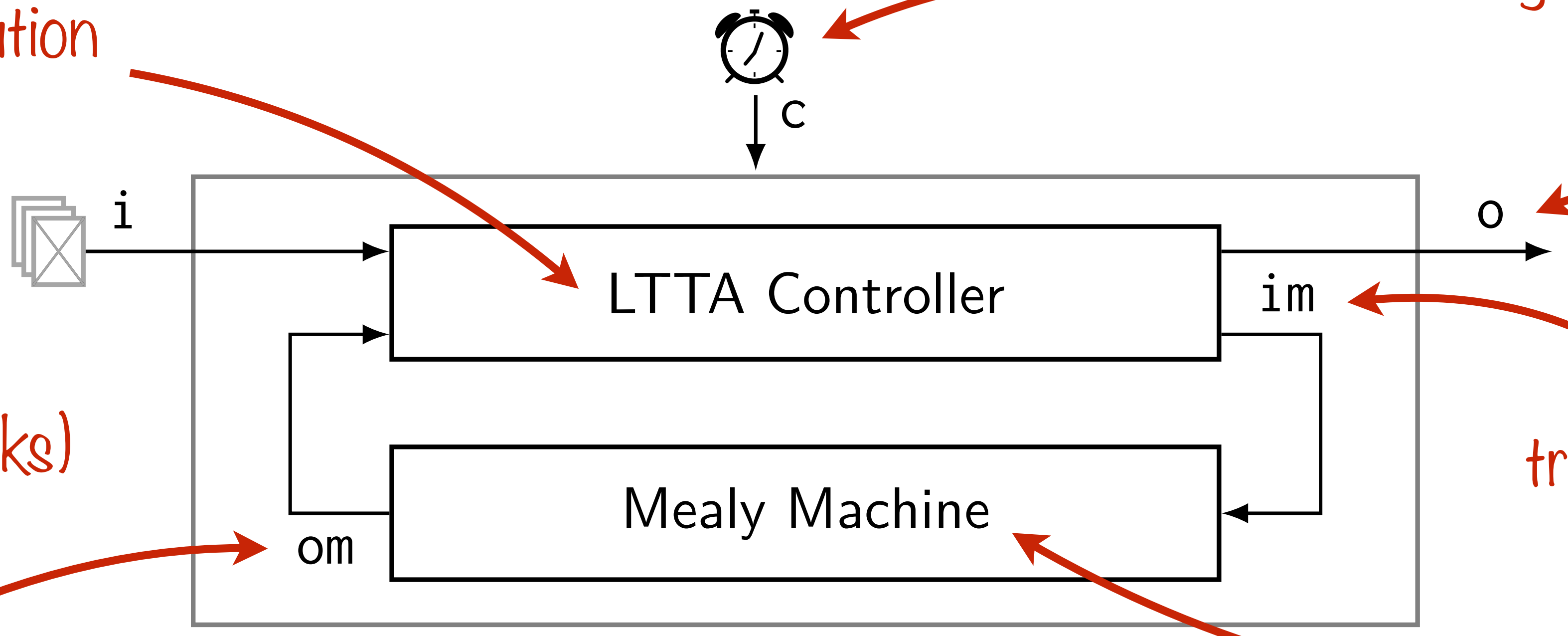
Synchronous application

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

Shell wrapper: Latency insensitive design (LID)
[Carloni, McMillan, Sangiovanni-Vincentelli]

A Synchronous Framework

A **middleware** controls the execution of the embedded application
The controller **waits** for new inputs and **delays** publications



Logical clock models node activation

Controls the execution of the application

Output

Input sampled from memories (links)

trigger application

Application output

Synchronous application

```
let node ltta_node(i) = o where
  rec (o, im) = ltta_controller(i, om)
```

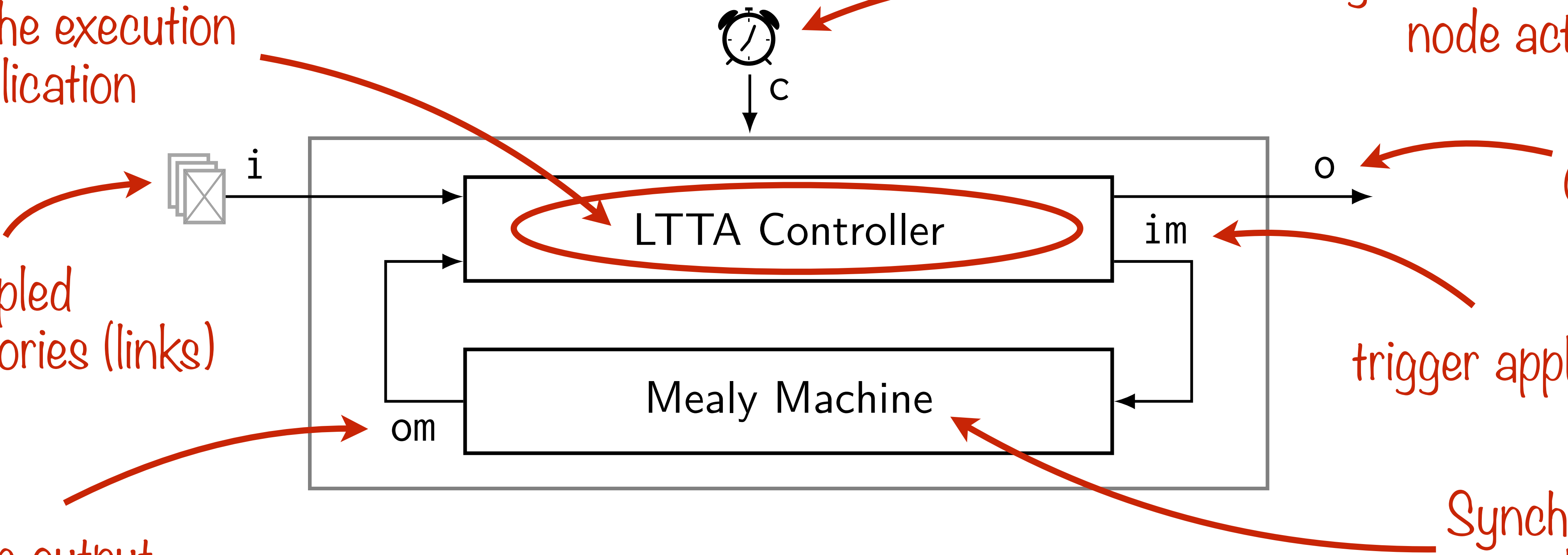
Controllers are synchronous programs too!

A Synchronous Framework

A **middleware** controls the execution of the embedded application
The controller **waits** for new inputs and **delays** publications

Controls the execution of the application

Logical clock models node activation



Input sampled from memories (links)

Output

trigger application

Application output

Synchronous application

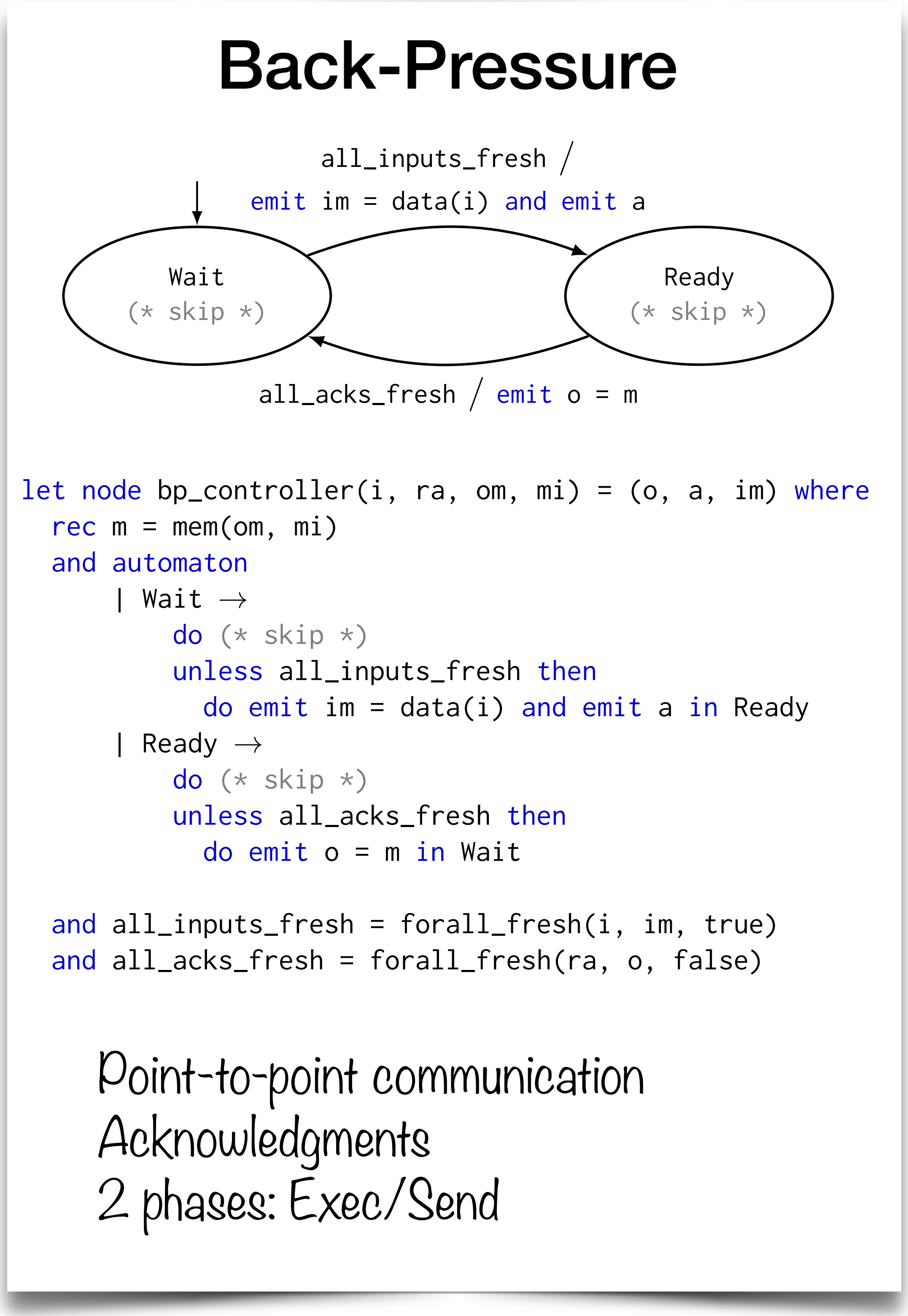
```
let node ltta_node(i) = o where
  rec (o, im) = ltta_controller(i, om)
```

Controllers are synchronous programs too!

The LTTA protocols

[TPB+08, CB08]

The LTTA protocols



Back-Pressure

```
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)
```

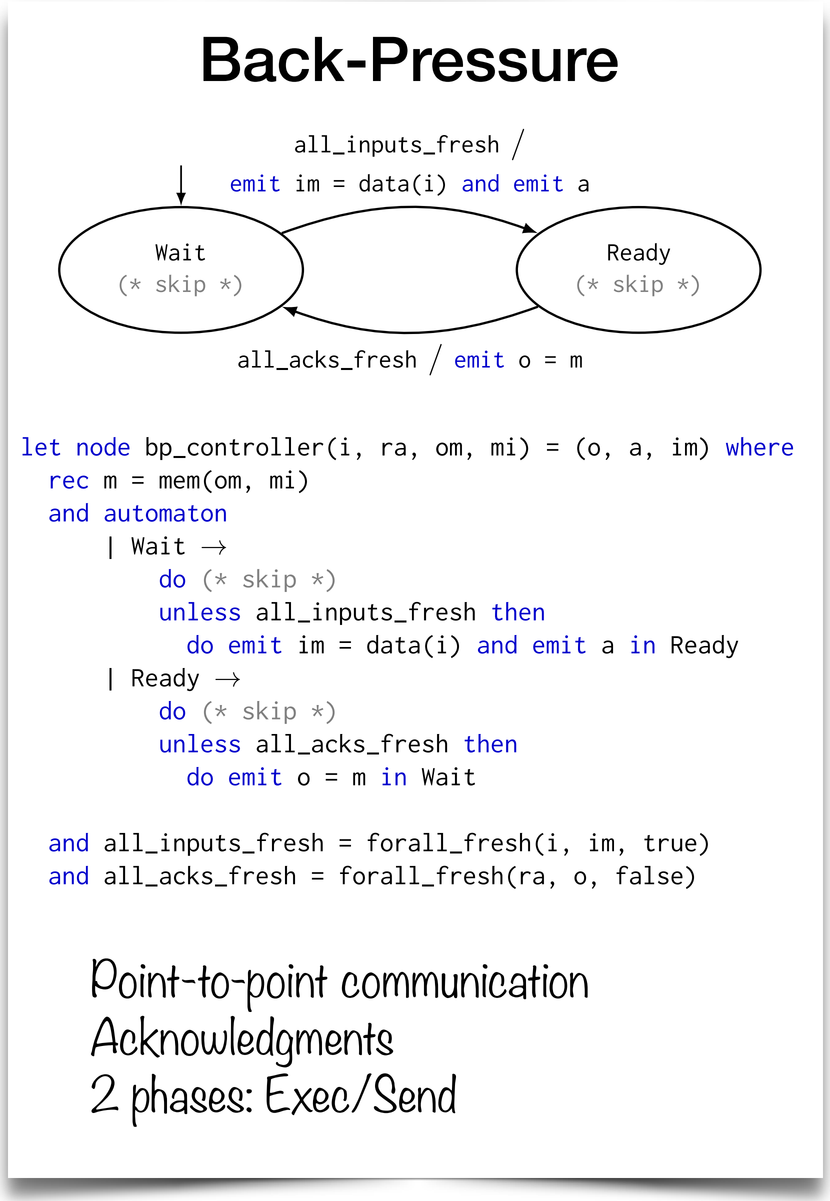
Point-to-point communication
Acknowledgments
2 phases: Exec/Send

Inspired by elastic circuits
[Cortadella, Kishinevsky, ...]

[Benveniste, Caspi, Di Natale, Pinello, Sangiovanni-Vincentelli, Tripakis]

[TPB+08, CB08]

The LTTA protocols



Back-Pressure

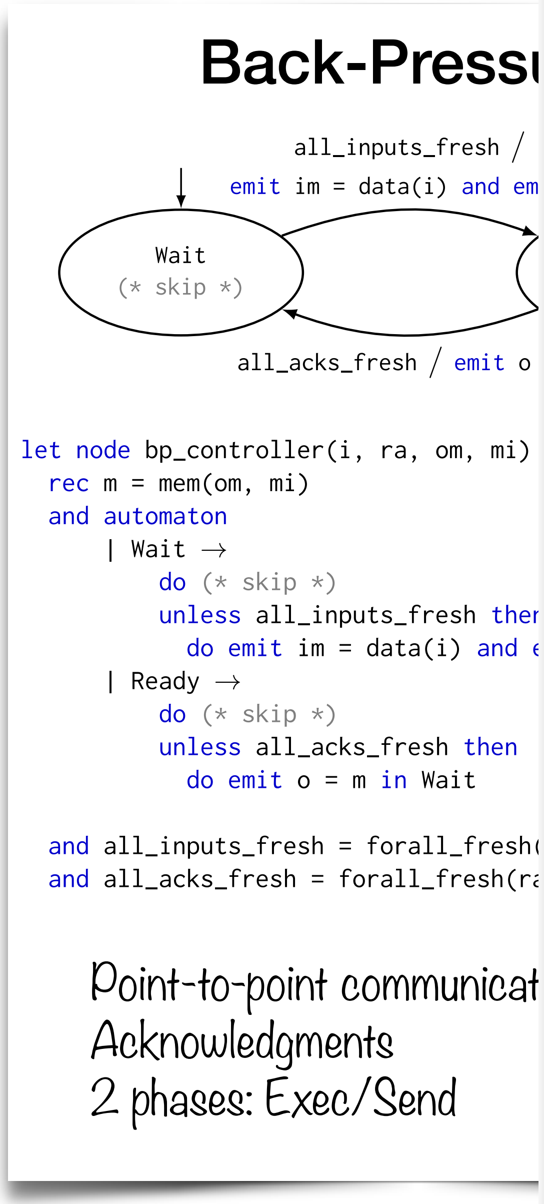
all_inputs_fresh /
emit im = data(i) and emit a
Wait
(* skip *)
Ready
(* skip *)
all_acks_fresh / emit o = m

```
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)
```

Point-to-point communication
Acknowledgments
2 phases: Exec/Send

[TPB+08, CB08]

The LTTA protocols

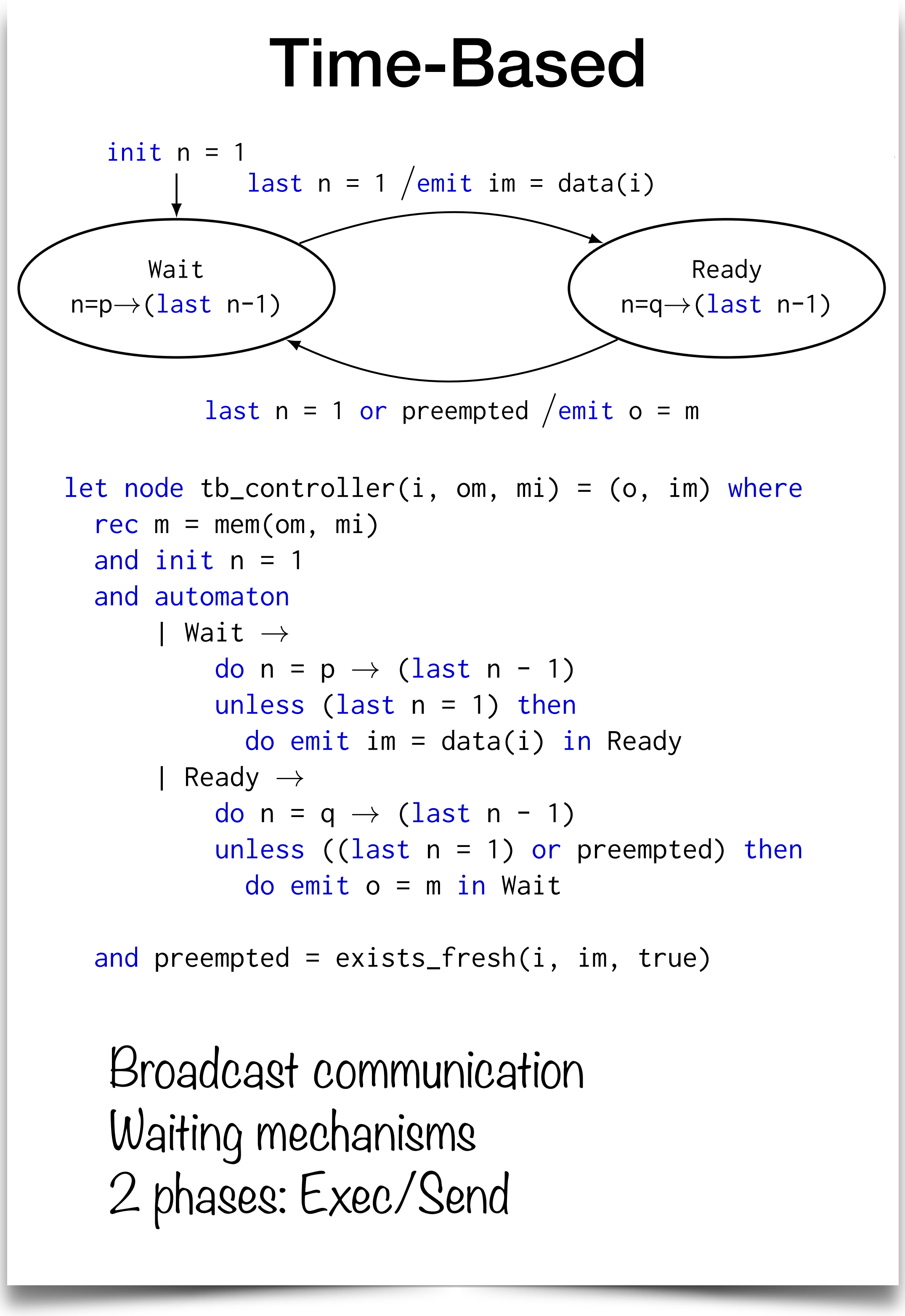


Back-Pressu

all_inputs_fresh /
emit im = data(i) and em
Wait
(* skip *)
all_acks_fresh / emit o

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

  and all_inputs_fresh = forall_fresh(
  and all_acks_fresh = forall_fresh(ra
```

Point-to-point communicat
Acknowledgments
2 phases: Exec/Send



Time-Based

init n = 1
last n = 1 /emit im = data(i)
Wait
n=p→(last n-1)
Ready
n=q→(last n-1)
last n = 1 or preempted /emit o = m

```
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)
```

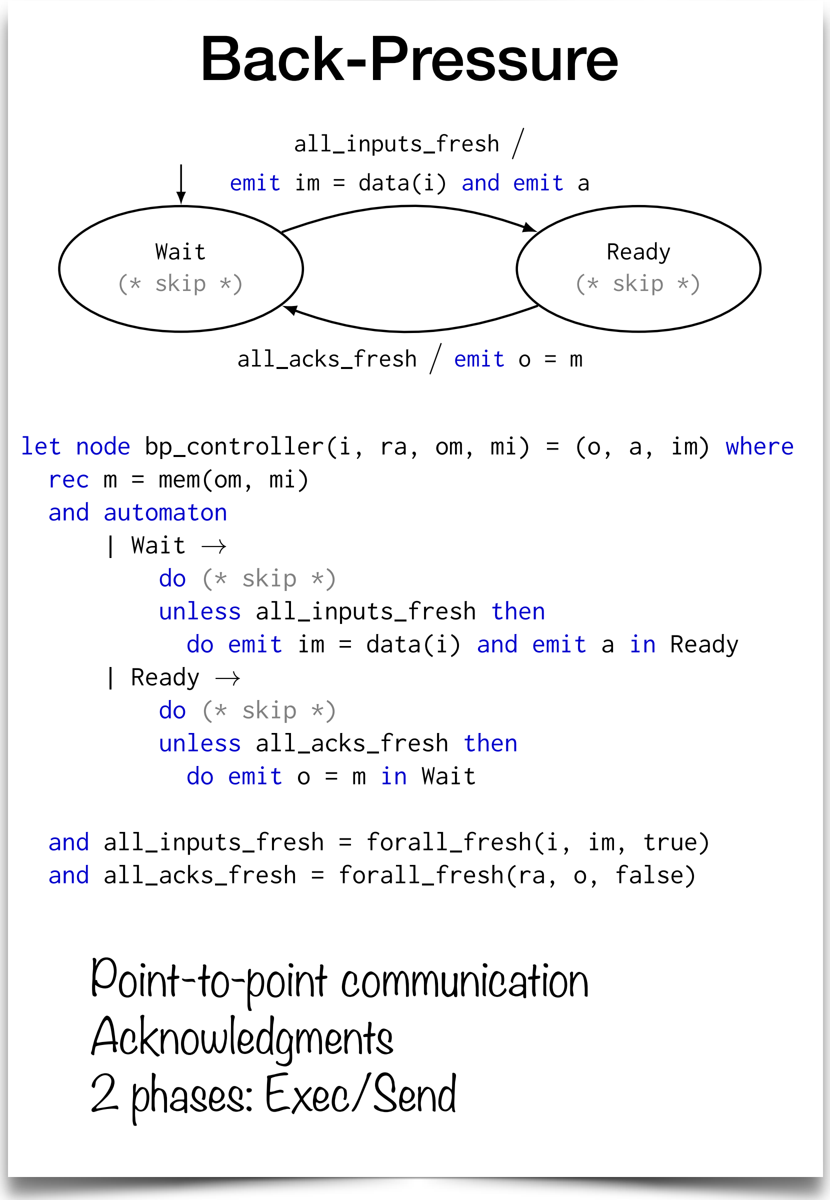
Broadcast communication
Waiting mechanisms
2 phases: Exec/Send

[Benveniste, Caspi]

Replace acknowledgments with timeouts

[TPB+08, CB08]

The LTTA protocols

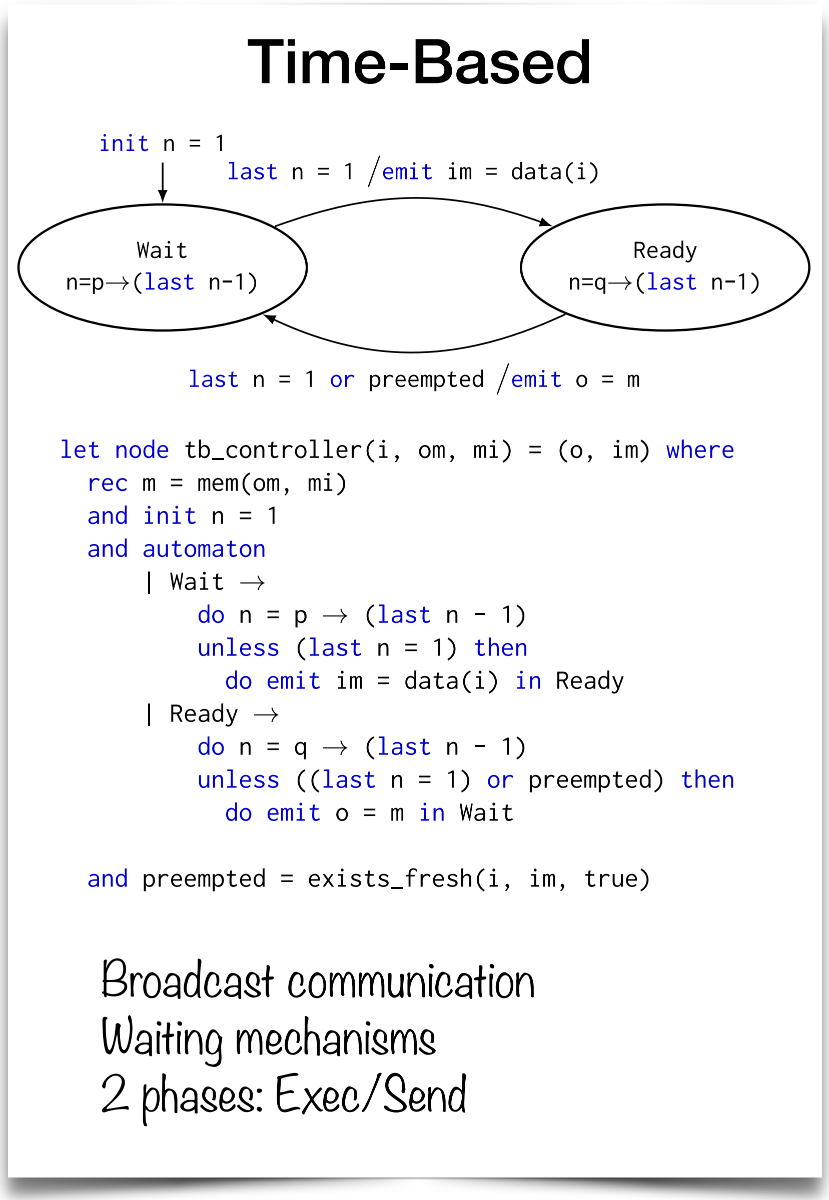


Back-Pressure

all_inputs_fresh /
emit im = data(i) and emit a
Wait
(* skip *)
Ready
(* skip *)
all_acks_fresh / emit o = m

```
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)
```

Point-to-point communication
Acknowledgments
2 phases: Exec/Send



Time-Based

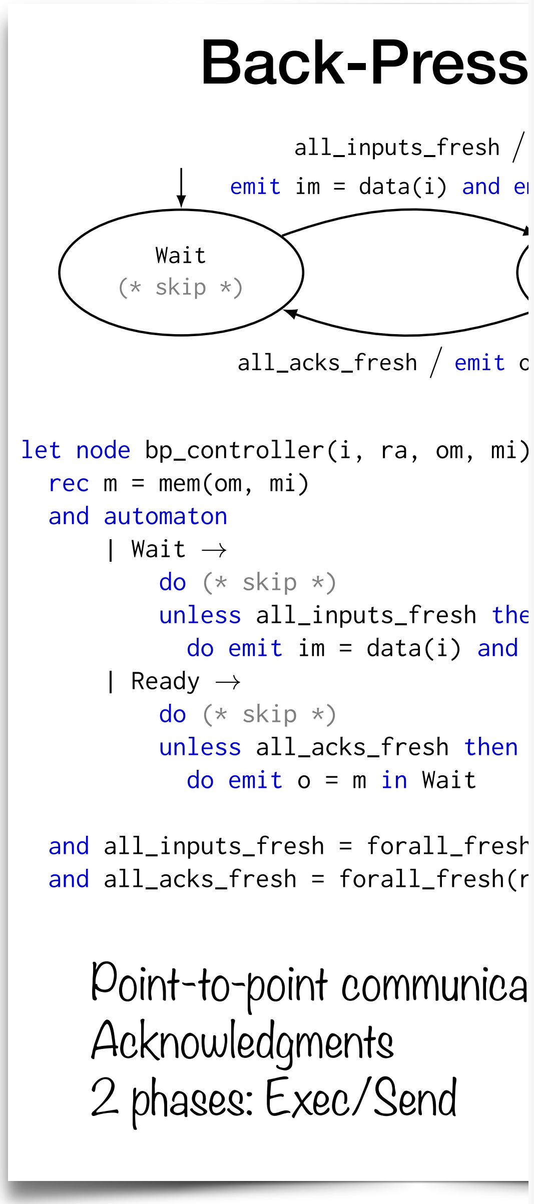
init n = 1
last n = 1 /emit im = data(i)
Wait
n=p→(last n-1)
Ready
n=q→(last n-1)
last n = 1 or preempted /emit o = m

```
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)
```

Broadcast communication
Waiting mechanisms
2 phases: Exec/Send

[TPB+08, CB08]

The LTTA protocols

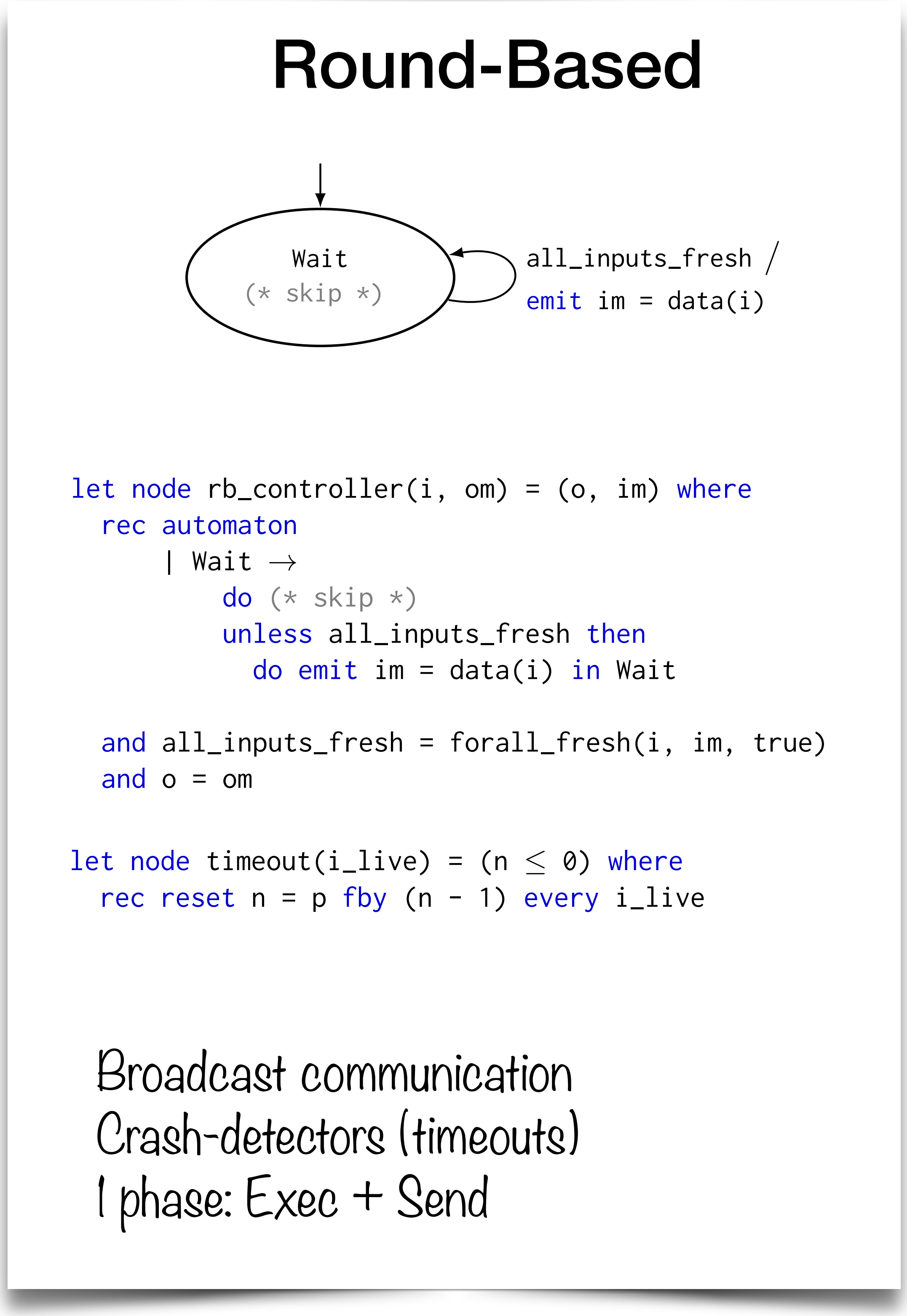


Back-Press

all_inputs_fresh /
emit im = data(i) and e
Wait
(* skip *)
all_acks_fresh / emit c

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

  and all_inputs_fresh = forall_fresh
  and all_acks_fresh = forall_fresh(r
```

Point-to-point communica
Acknowledgments
2 phases: Exec/Send



Round-Based

Wait
(* skip *)
all_inputs_fresh /
emit im = data(i)

```
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

let node timeout(i_live) = (n ≤ 0) where
  rec reset n = p fby (n - 1) every i_live
```

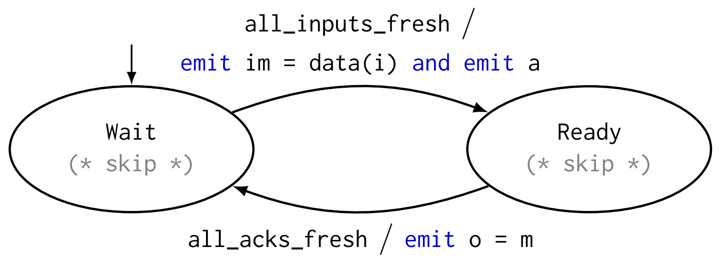
Broadcast communication
Crash-detectors (timeouts)
1 phase: Exec + Send

Inspired by
distributed algorithms
[Attiya, Dwork, Lynch, ...]

[TPB+08, CB08]

The LTTA protocols

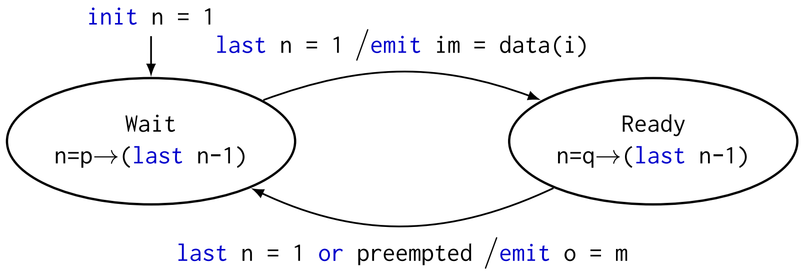
Back-Pressure



```
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)
```

Point-to-point communication
Acknowledgments
2 phases: Exec/Send

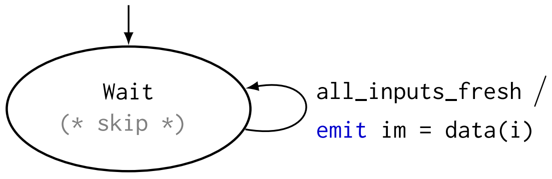
Time-Based


init n = 1
last n = 1 /emit im = data(i)
Wait
n=p→(last n-1)
Ready
n=q→(last n-1)
last n = 1 or preempted /emit o = m

```
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)
```

Broadcast communication
Waiting mechanisms
2 phases: Exec/Send

Round-Based



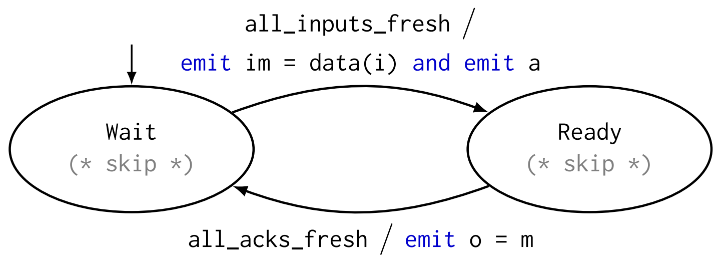
```
let node rb_controller(i, om) = (o, im) where
  rec automaton
      | Wait →
          do (* skip *)
          unless all_inputs_fresh then
            do emit im = data(i) in Wait

  and all_inputs_fresh = forall_fresh(i, im, true)
  and o = om

let node timeout(i_live) = (n ≤ 0) where
  rec reset n = p fby (n - 1) every i_live
```

Broadcast communication
Crash-detectors (timeouts)
1 phase: Exec + Send

[TPB+08, CB08]

The LTTA protocols

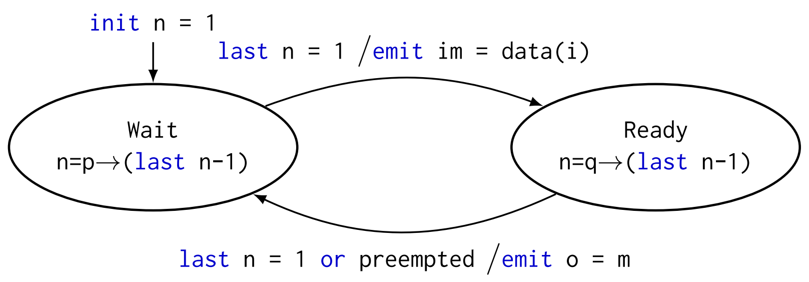
Back-Pressure



```
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)
```

Point-to-point communication
Acknowledgments
2 phases: Exec/Send

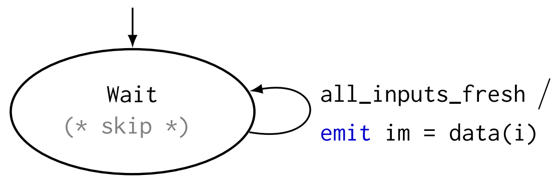
Time-Based


init n = 1
last n = 1 /emit im = data(i)
Wait
n=p→(last n-1)
Ready
n=q→(last n-1)
last n = 1 or preempted /emit o = m

```
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)
```

Broadcast communication
Waiting mechanisms
2 phases: Exec/Send

Round-Based



```
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

let node timeout(i_live) = (n ≤ 0) where
  rec reset n = p fby (n - 1) every i_live
```

Broadcast communication
Crash-detectors (timeouts)
1 phase: Exec + Send

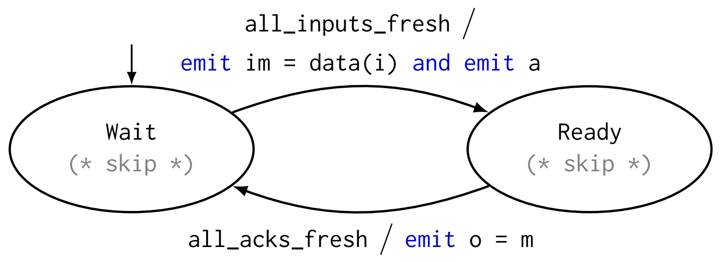
Architecture independent

Block if a node crashes

[TPB+08, CB08]

The LTTA protocols

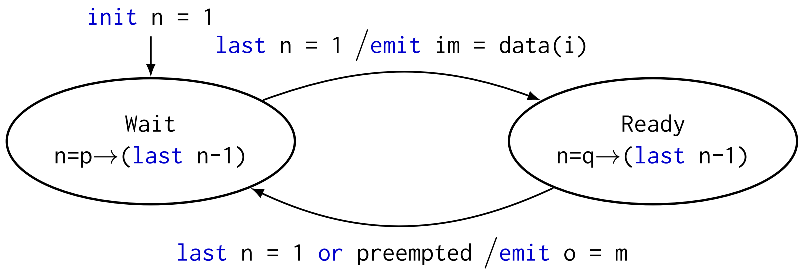
Back-Pressure



```
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)
```

Point-to-point communication
Acknowledgments
2 phases: Exec/Send

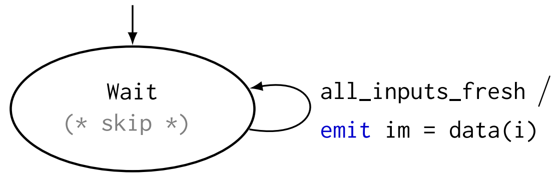
Time-Based


init n = 1
last n = 1 /emit im = data(i)
Wait
n=p→(last n-1)
Ready
n=q→(last n-1)
last n = 1 or preempted /emit o = m

```
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)
```

Broadcast communication
Waiting mechanisms
2 phases: Exec/Send

Round-Based



```
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

let node timeout(i_live) = (n ≤ 0) where
  rec reset n = p fby (n - 1) every i_live
```

Broadcast communication
Crash-detectors (timeouts)
1 phase: Exec + Send

Architecture independent

Block if a node crashes

Require timing characteristics

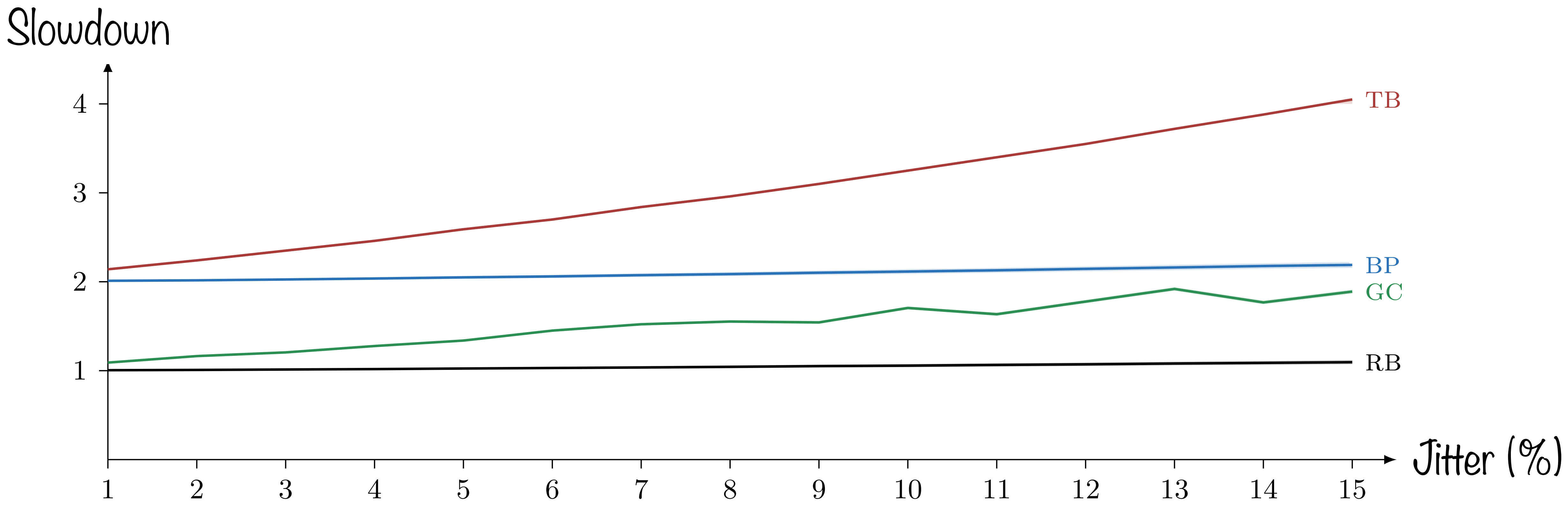
Can run in degraded mode

[TPB+08, CB08]

Comparisons with clock synchronization

Zélus simulations of the FGS example
Compute slowdown compared to a synchronous execution*

Execution period << Communication delay



Global Clock: based on a master clock synchronization [Kopetz]
`arbitrary(t_min, t_max)`: Random choice

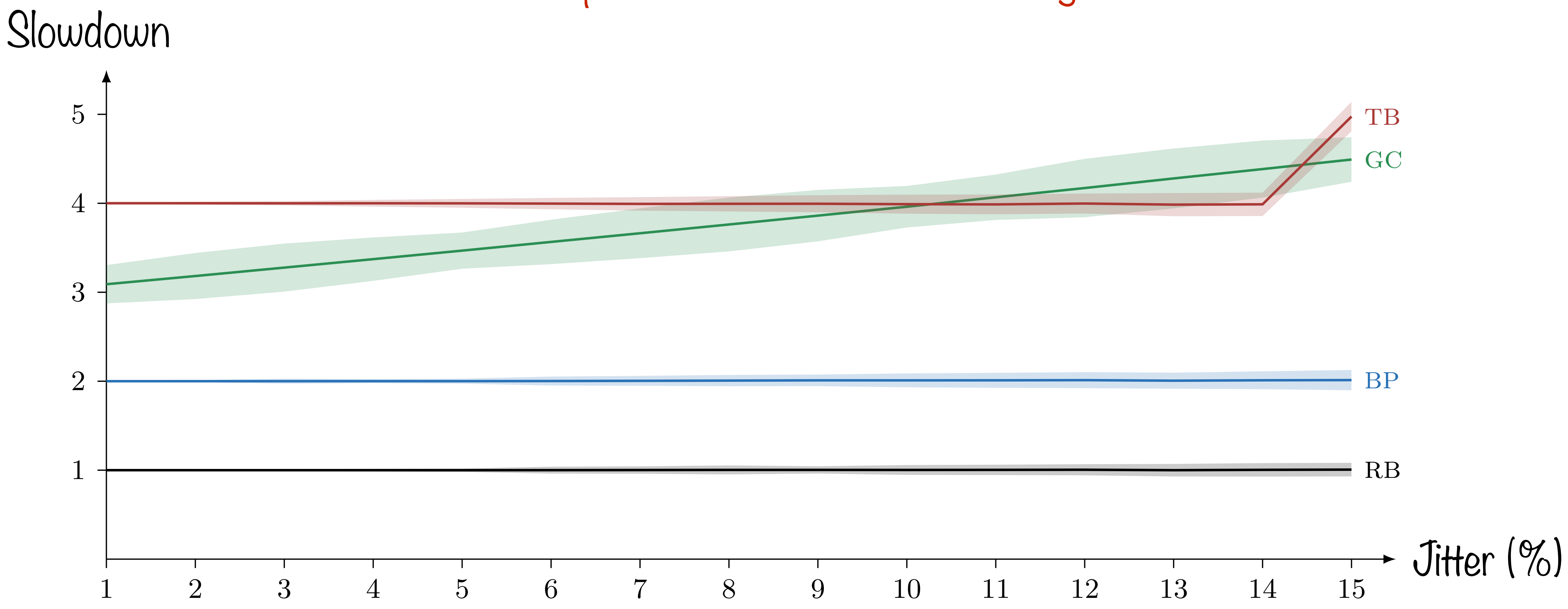
BP: Back-Pressure
TB: Time-Based
RB: Round-Based
GC: Global Clock

*The smaller, the better

Comparisons with clock synchronization

Zélus simulations of the FGS example
Compute slowdown compared to a synchronous execution*

Execution period >> Communication delay



Global Clock: based on a master clock synchronization [Kopetz]
`arbitrary(t_min, t_max)`: Random choice

BP: Back-Pressure
TB: Time-Based
RB: Round-Based
GC: Global Clock

*The smaller, the better

Summary

Loosely Time-Triggered Architectures:
How to deploy synchronous code?
Add a layer of middleware
Three protocols

Contributions:

- Unified synchronous framework
- Executable specifications
- Correctness proofs
- Optimization and comparisons

Loosely Time-Triggered Architectures:
Improvements and Comparisons

A

Loosely Time-Triggered Architectures:
Improvements and Comparisons

GUILLAUME BAUDART, École normale supérieure
ALBERT BENVENISTE, INRIA, Rennes
TIMOTHY BOURKE, INRIA, Paris

1. INTRODUCTION

LTTA are lightweight protocols to ensure the correct execution of synchronous code running on a quasi-periodic architecture

[EMSOFT 2015, ACM TECS 2016]

Overview

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

[BP13, BDL06]

Overview

How to simulate
constrained nondeterminism?

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

[BP13, BDL06]

Overview

How to simulate constrained nondeterminism?

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

Zélus
Synchronous language
Continuous + Discrete
Modular compilation
Numeric solver
[Benveniste, Bourke, Caillaud, Pouzet]

Uppaal
Timed automata
Nondeterminism
Symbolic representation
[Behrmann, David, Larsen,...]

[BP13, BDL06]

Overview

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

Contributions

Zélus extended with timed nondeterminism

Symbolic simulation

Modular source-to-source compilation

Prototype implementation

How to simulate constrained nondeterminism?

Zélus
Synchronous language
Continuous + Discrete
Modular compilation
Numeric solver
[Benveniste, Bourke, Caillaud, Pouzet]

Uppaal
Timed automata
Nondeterminism
Symbolic representation
[Behrmann, David, Larsen,...]

[BP13, BDL06]

Timed Nondeterminism

Simulate both the embedded **application** and the **architecture**

Zélus: mix discrete-time and continuous-time dynamics expressed with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

Embedded application activates on signal emissions

Timed Nondeterminism

Simulate both the embedded **application** and the **architecture**

Zélus: mix discrete-time and continuous-time dynamics expressed with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

We only use der x = 1

Embedded application activates on signal emissions

Timed Nondeterminism

Simulate both the embedded **application** and the **architecture**

Zélus: mix discrete-time and continuous-time dynamics expressed with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

We only use der x = 1

Embedded application activates on signal emissions

Zsy: Zélus extended with timed nondeterminism

```
let hybrid metro(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}
```

Timed Nondeterminism

Simulate both the embedded **application** and the **architecture**

Zélus: mix discrete-time and continuous-time dynamics expressed with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

We only use der x = 1

Embedded application activates on signal emissions

Zsy: Zélus extended with timed nondeterminism

```
let hybrid metro(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}
```

timer (time elapsing)

Timed Nondeterminism

Simulate both the embedded **application** and the **architecture**

Zélus: mix discrete-time and continuous-time dynamics expressed with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

We only use der x = 1

Embedded application activates on signal emissions

Zsy: Zélus extended with timed nondeterminism

```
let hybrid metro(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}
```

timer (time elapsing)

guard (may)

Timed Nondeterminism

Simulate both the embedded **application** and the **architecture**

Zélus: mix discrete-time and continuous-time dynamics expressed with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z = up(x)
  and present z → do emit c done
```

We only use der x = 1

Embedded application activates on signal emissions

Zsy: Zélus extended with timed nondeterminism

```
let hybrid metro(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}
```

timer (time elapsing)

guard (may)

invariant (must)

Timed Nondeterminism

Simulate both the embedded **application** and the **architecture**

Zélus: mix discrete-time and continuous-time dynamics expressed with ODEs

```
let hybrid metro(t_min, t_max) = c where
  rec der x = 1.0 init -. arbitrary(t_min, t_max)
      reset z → -. arbitrary(t_min, t_max)
  and z =
  and pre
```

der x = 1

How to simulate such programs?

Zsy: Zélus extended with timed nondeterminism

```
let hybrid metro(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}
```

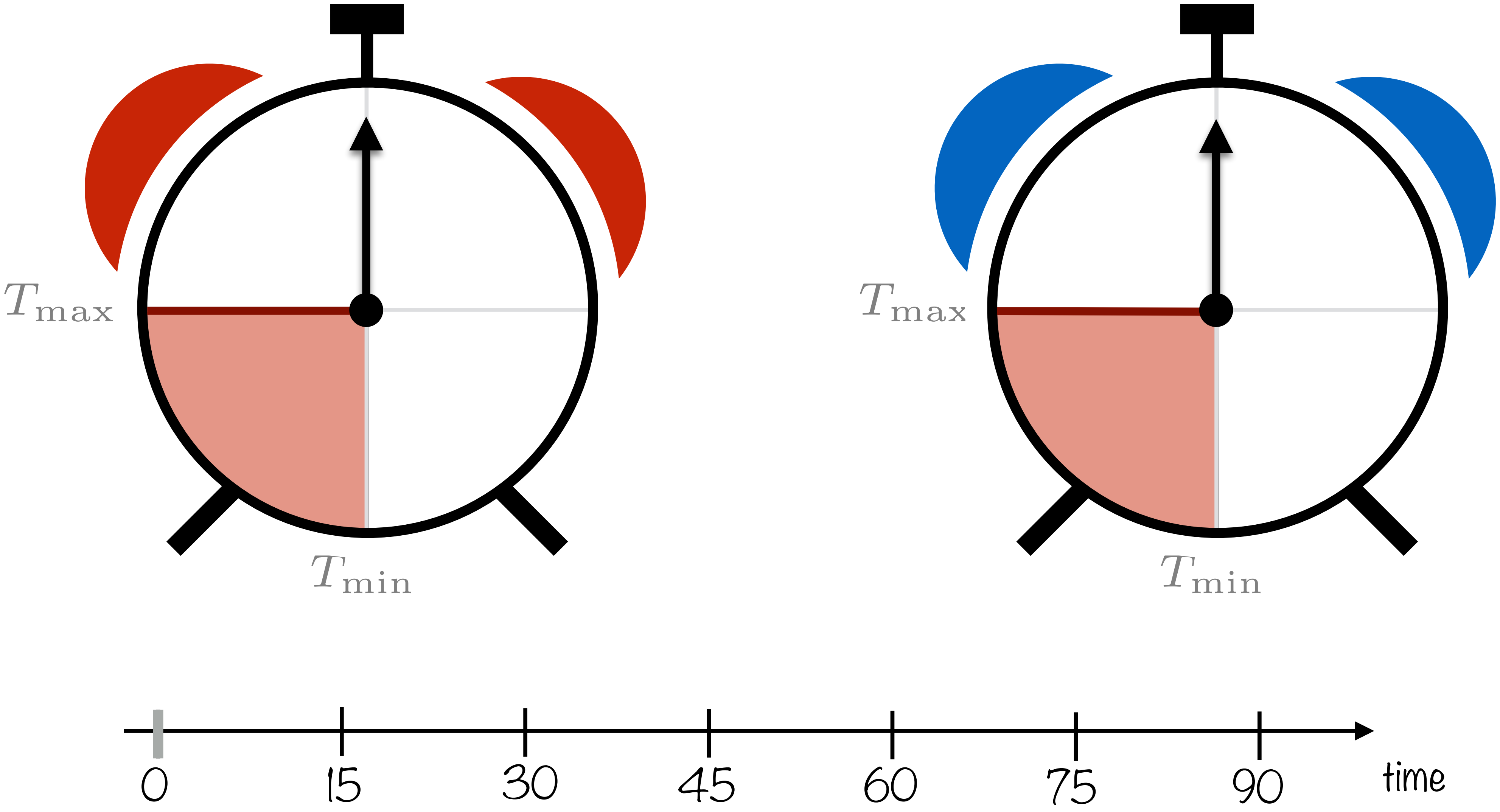
timer (time elapsing)

guard (may)

invariant (must)

Concrete Simulation

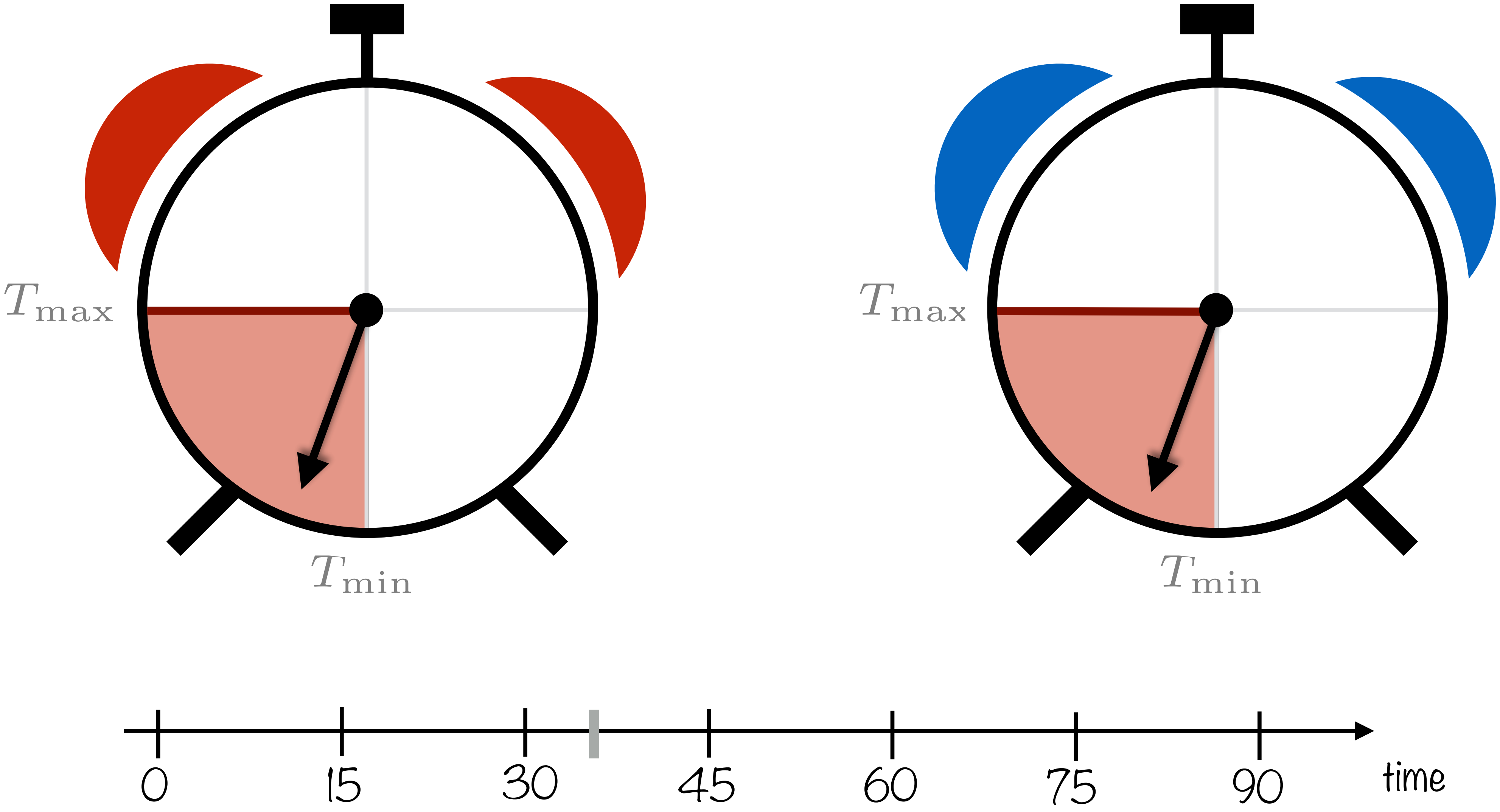
Example: 2-node quasi-periodic architecture



Random testing: test one execution, using numerical solvers

Concrete Simulation

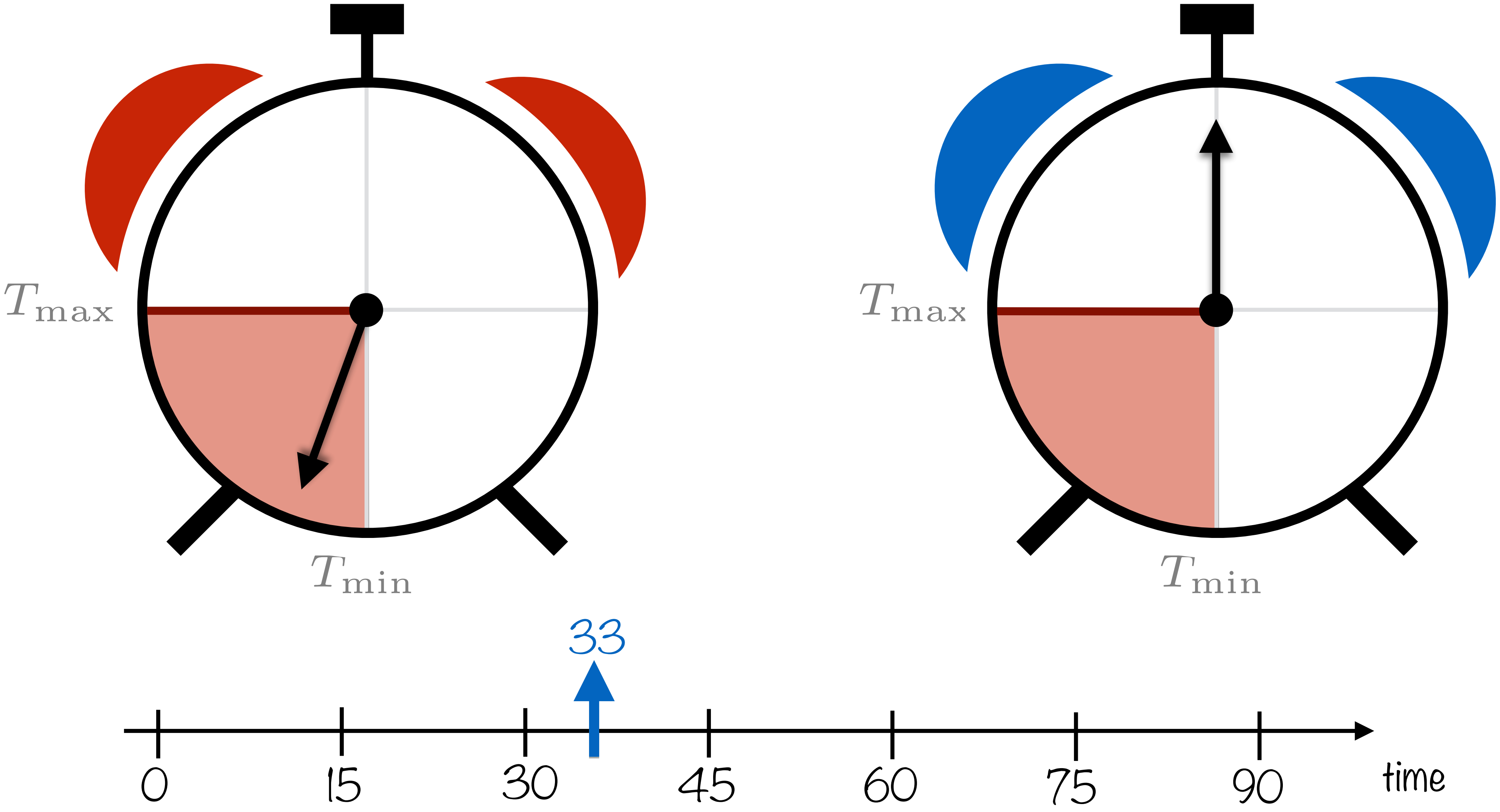
Example: 2-node quasi-periodic architecture



Random testing: test one execution, using numerical solvers

Concrete Simulation

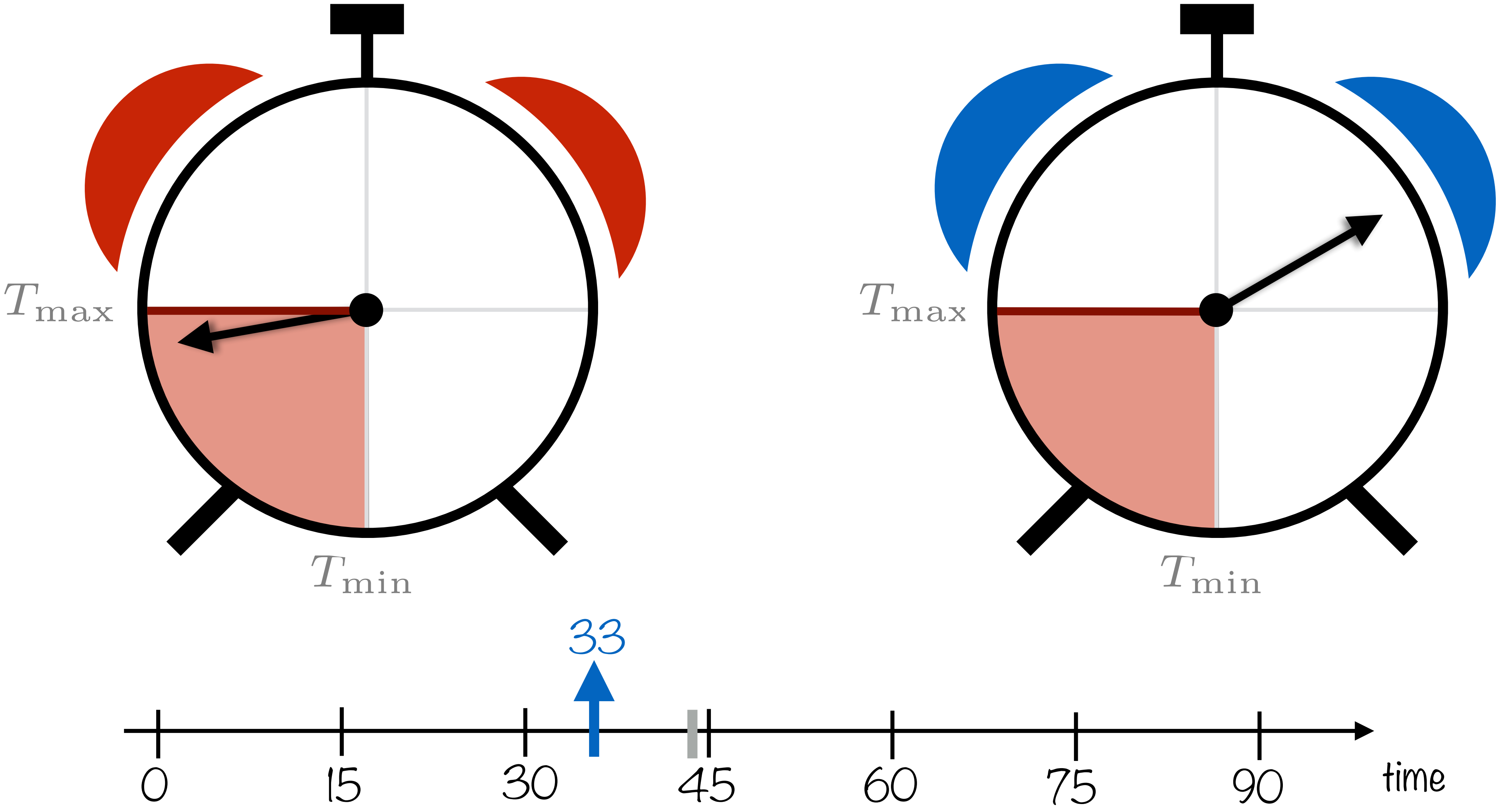
Example: 2-node quasi-periodic architecture



Random testing: test one execution, using numerical solvers

Concrete Simulation

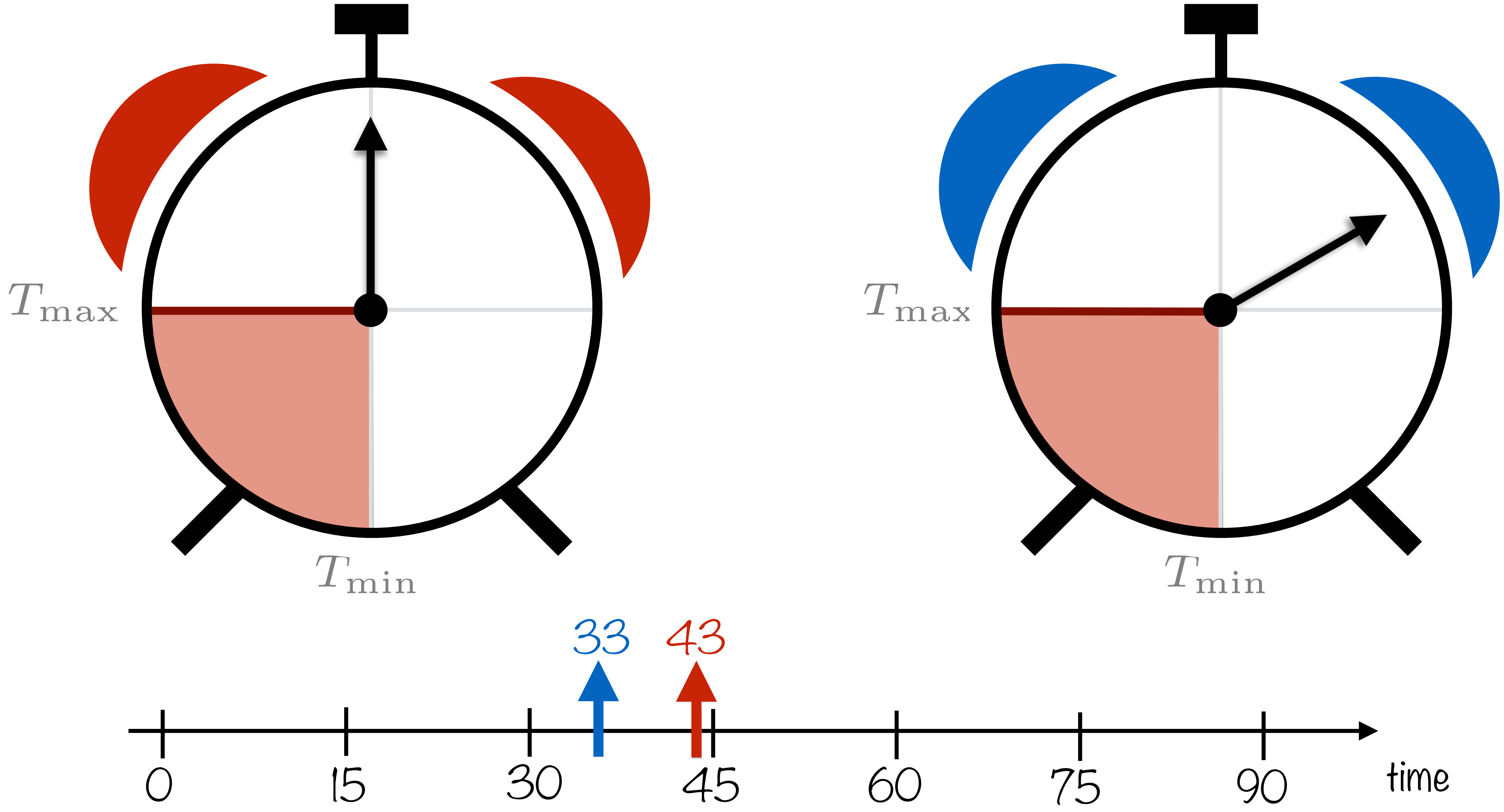
Example: 2-node quasi-periodic architecture



Random testing: test one execution, using numerical solvers

Concrete Simulation

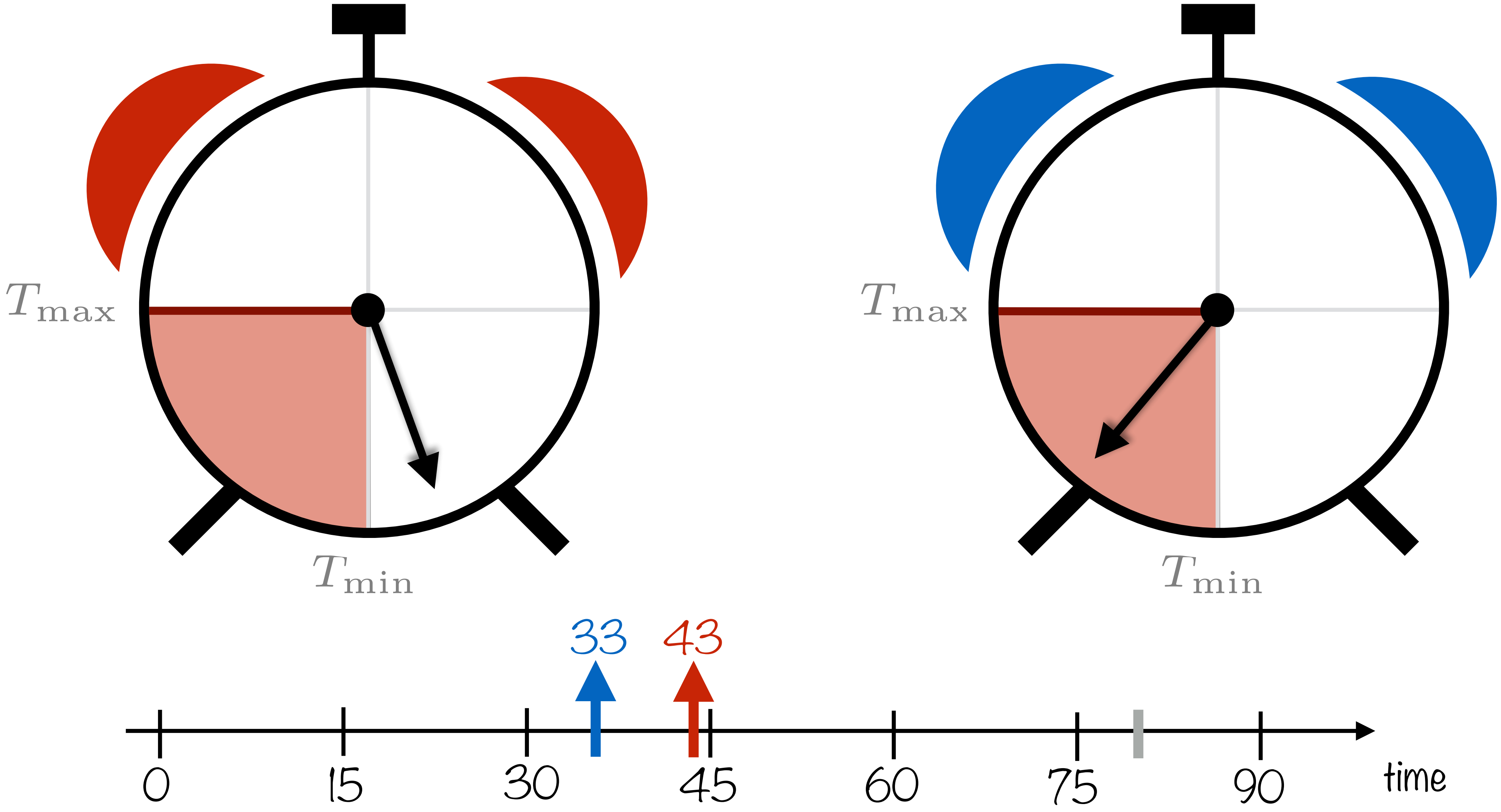
Example: 2-node quasi-periodic architecture



Random testing: test one execution, using numerical solvers

Concrete Simulation

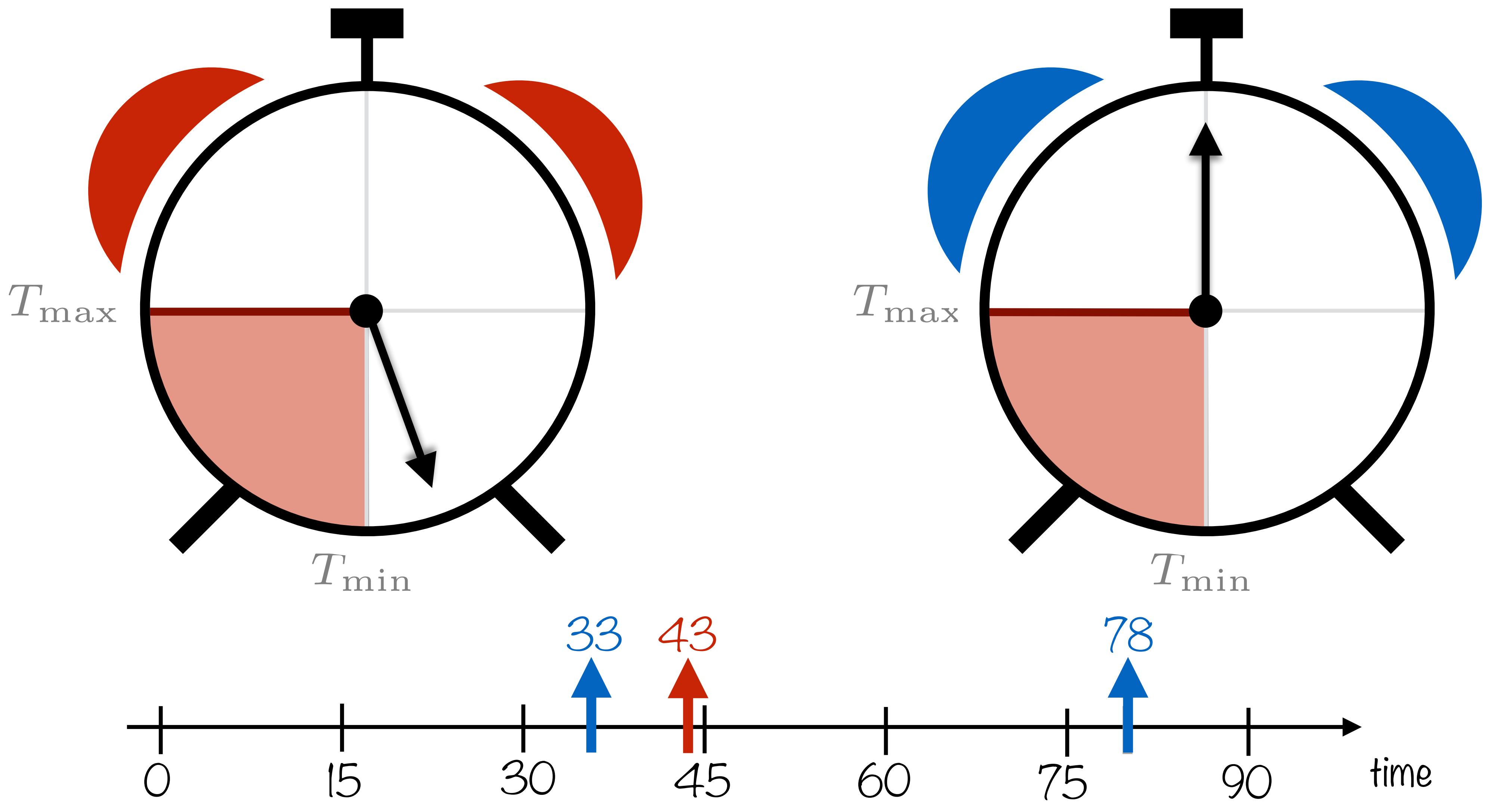
Example: 2-node quasi-periodic architecture



Random testing: test one execution, using numerical solvers

Concrete Simulation

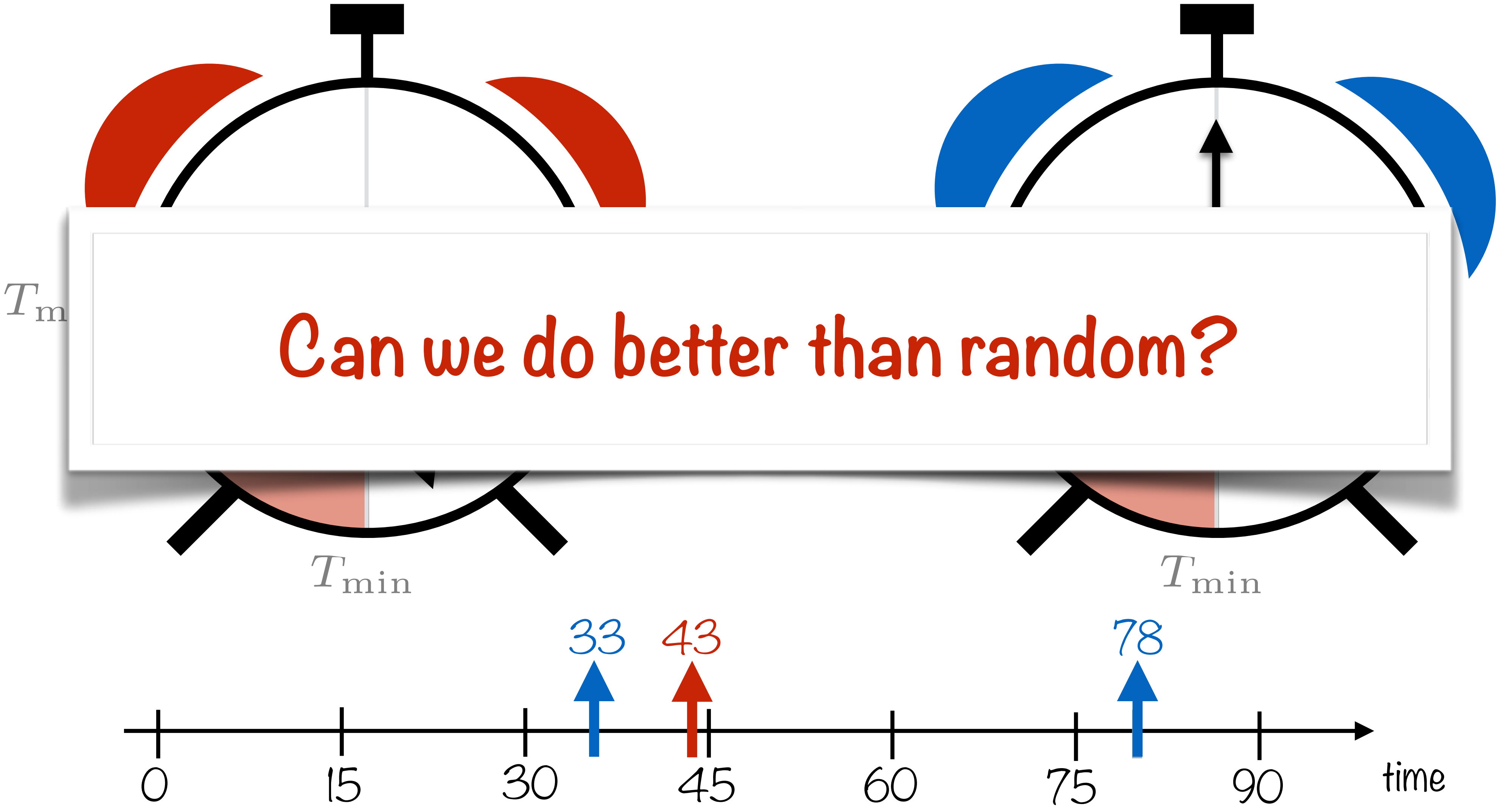
Example: 2-node quasi-periodic architecture



Random testing: test one execution, using numerical solvers

Concrete Simulation

Example: 2-node quasi-periodic architecture

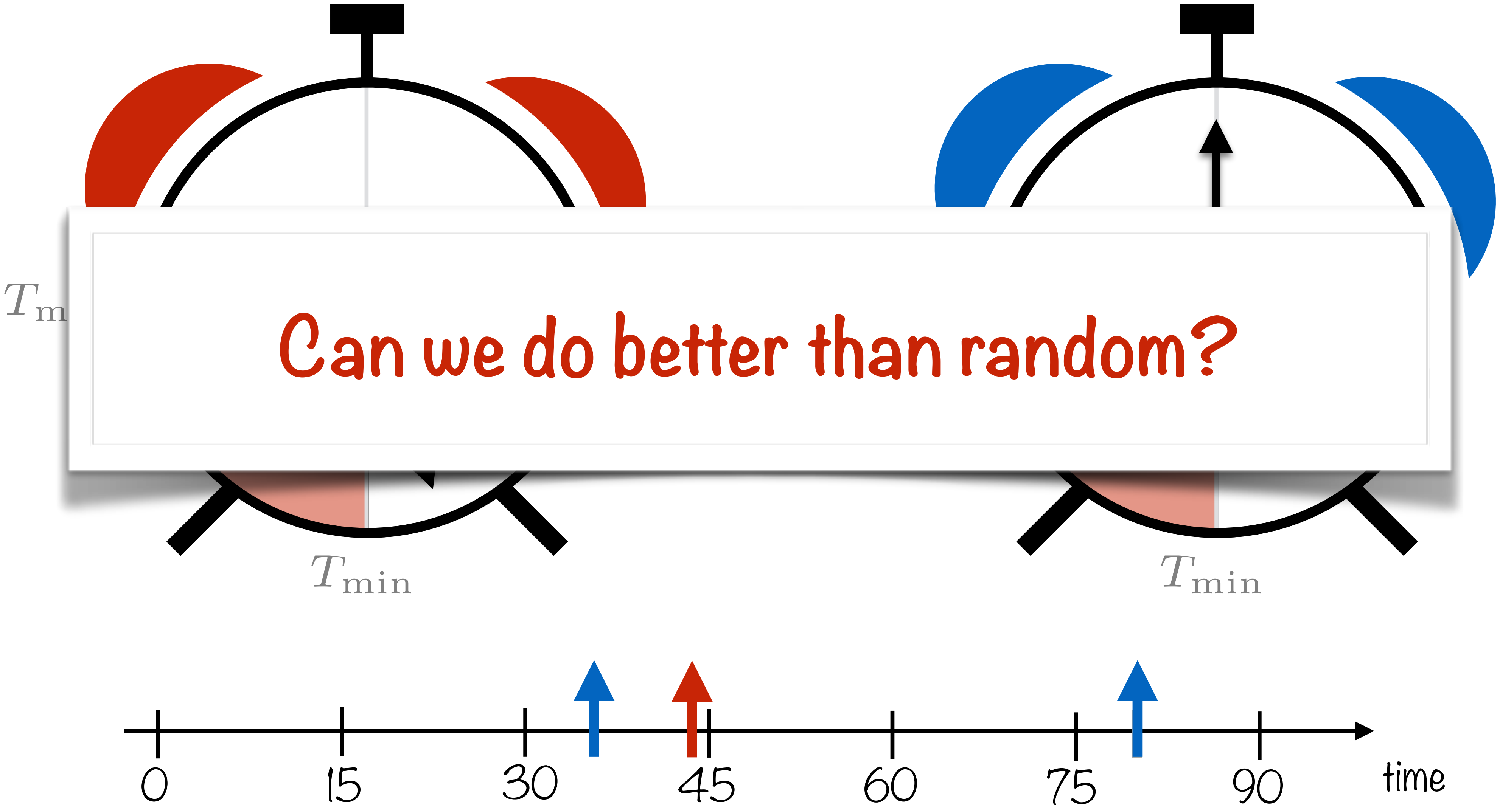


Can we do better than random?

Random testing: test one execution, using numerical solvers

Concrete Simulation

Example: 2-node quasi-periodic architecture

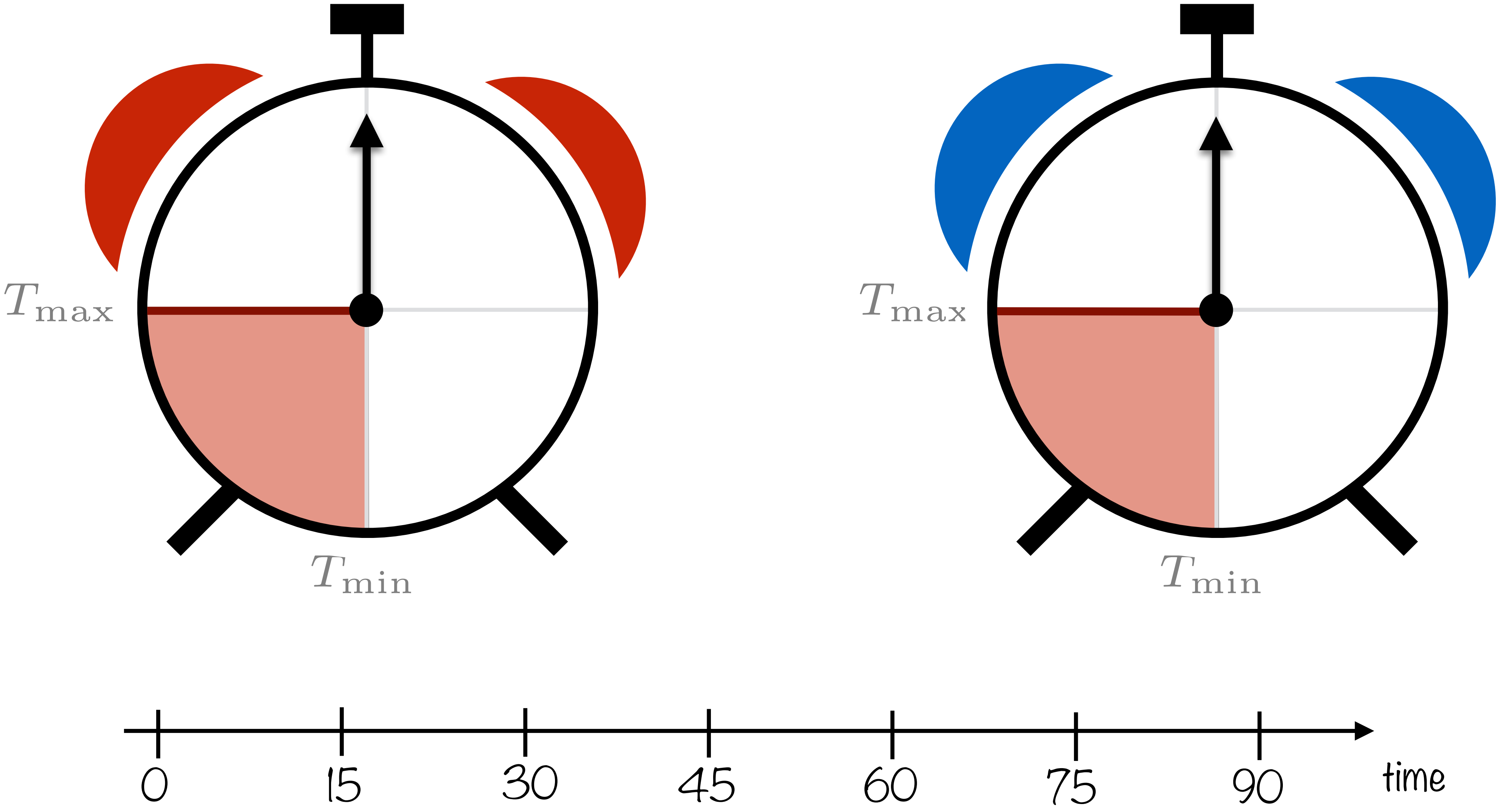


Can we do better than random?

Random testing: test one execution, using numerical solvers

Symbolic Simulation

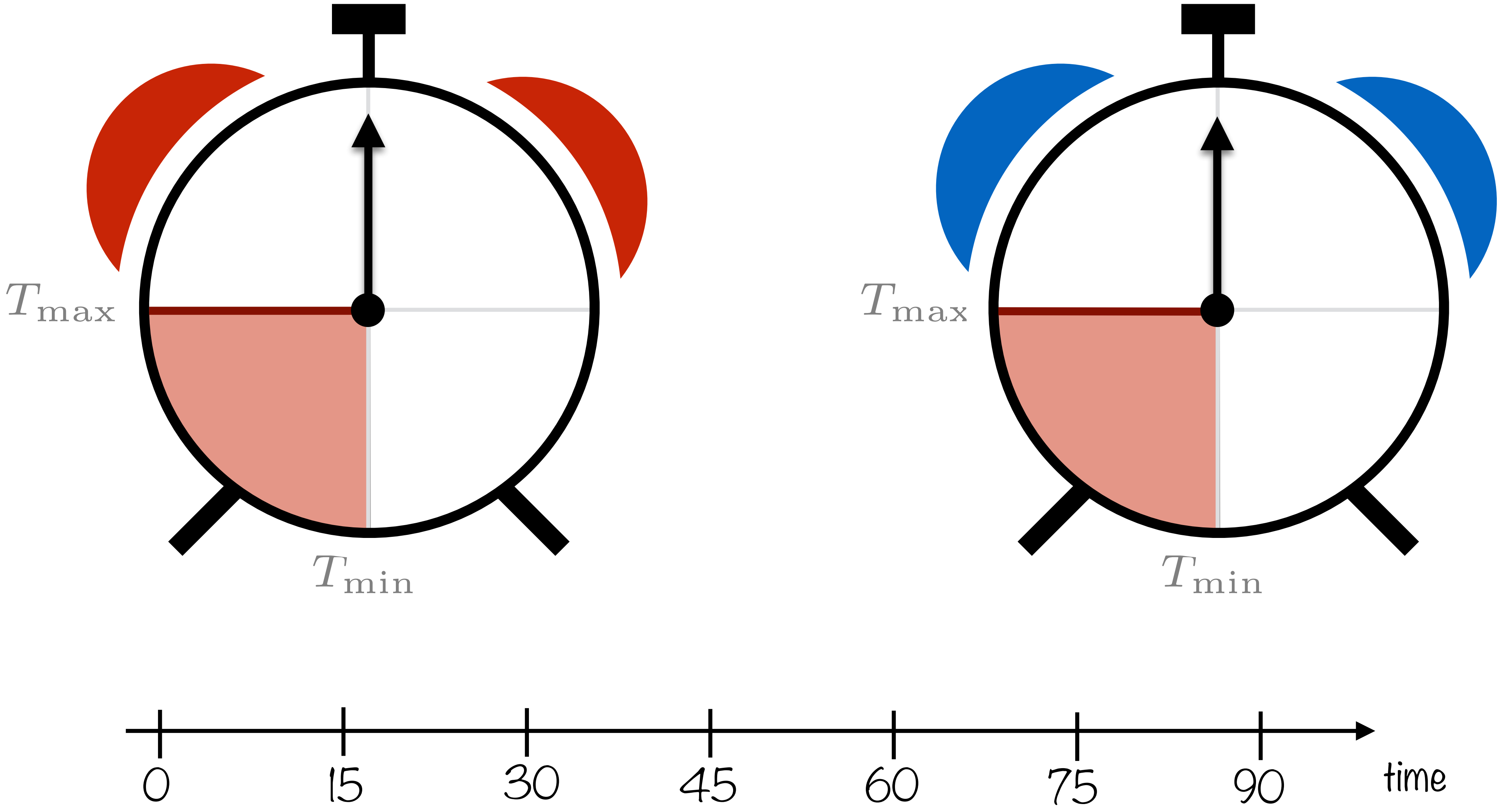
Example: a 2-node quasi-periodic architecture



Symbolic simulation: capture multiple executions, using DBMs

Symbolic Simulation

Example: a 2-node quasi-periodic architecture

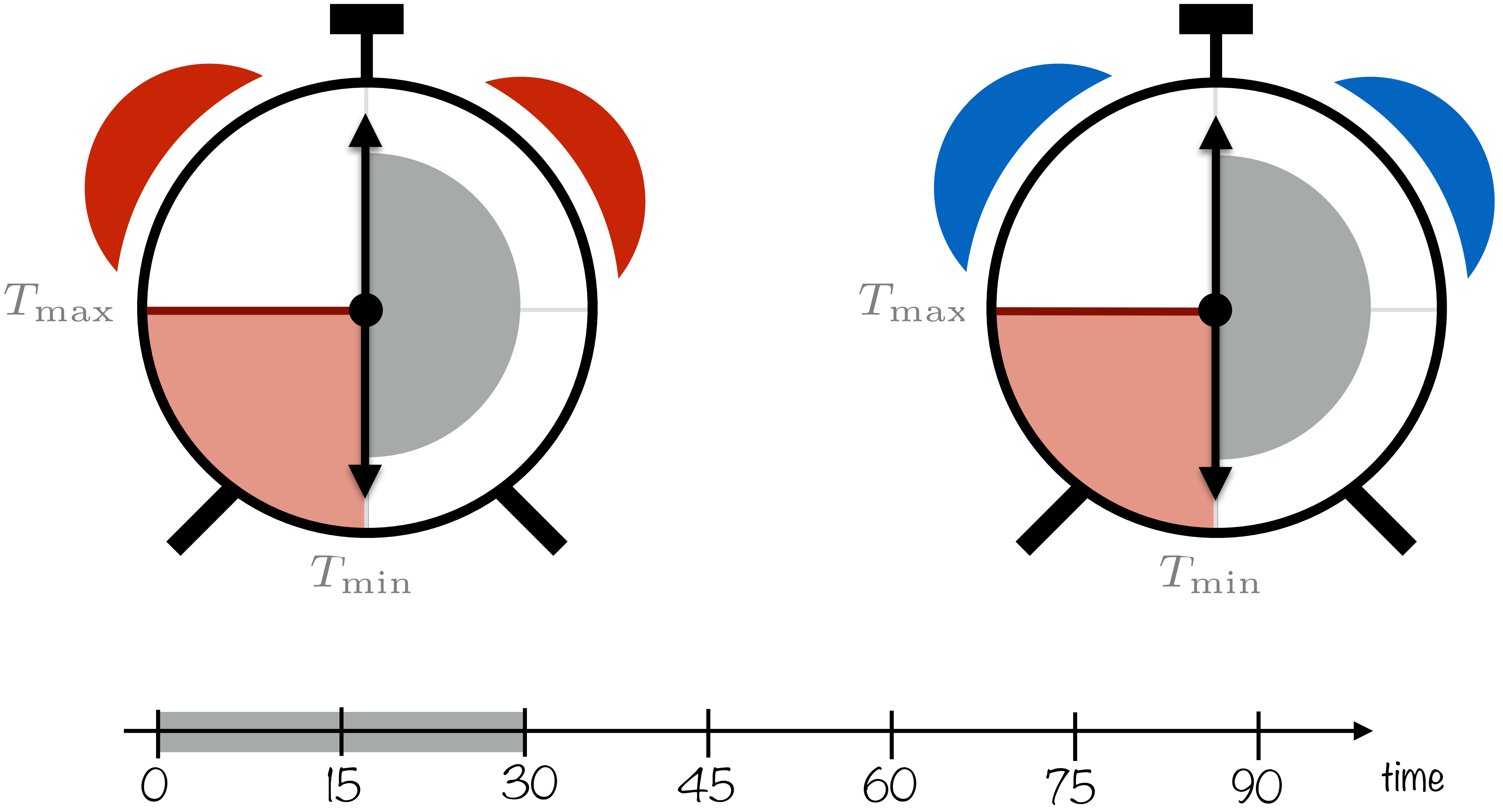


Symbolic simulation: capture multiple executions, using DBMs

Zones characterized by a set of possible choices

Symbolic Simulation

Example: a 2-node quasi-periodic architecture

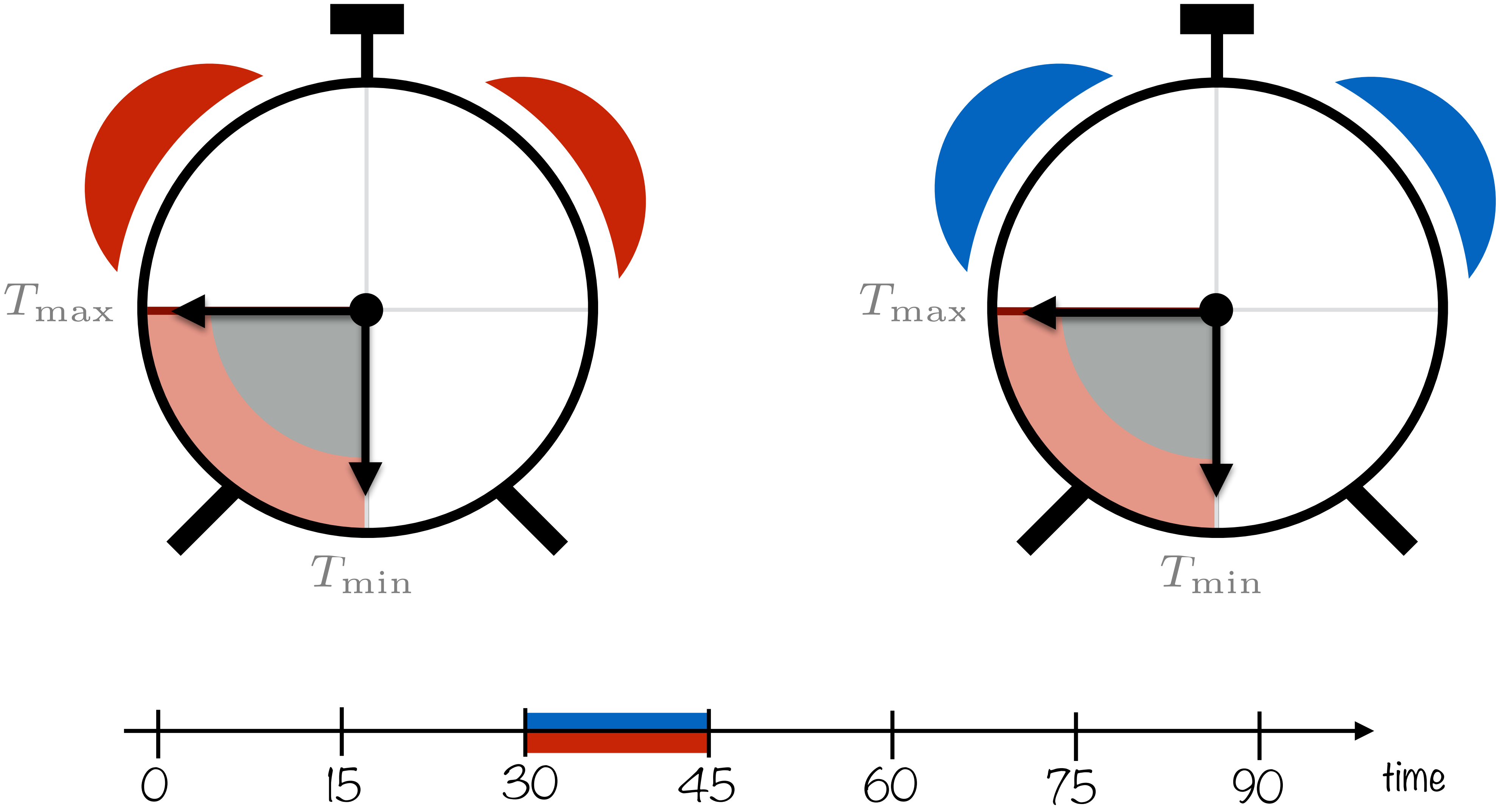


Symbolic simulation: capture multiple executions, using DBMs

Zones characterized by a set of possible choices

Symbolic Simulation

Example: a 2-node quasi-periodic architecture

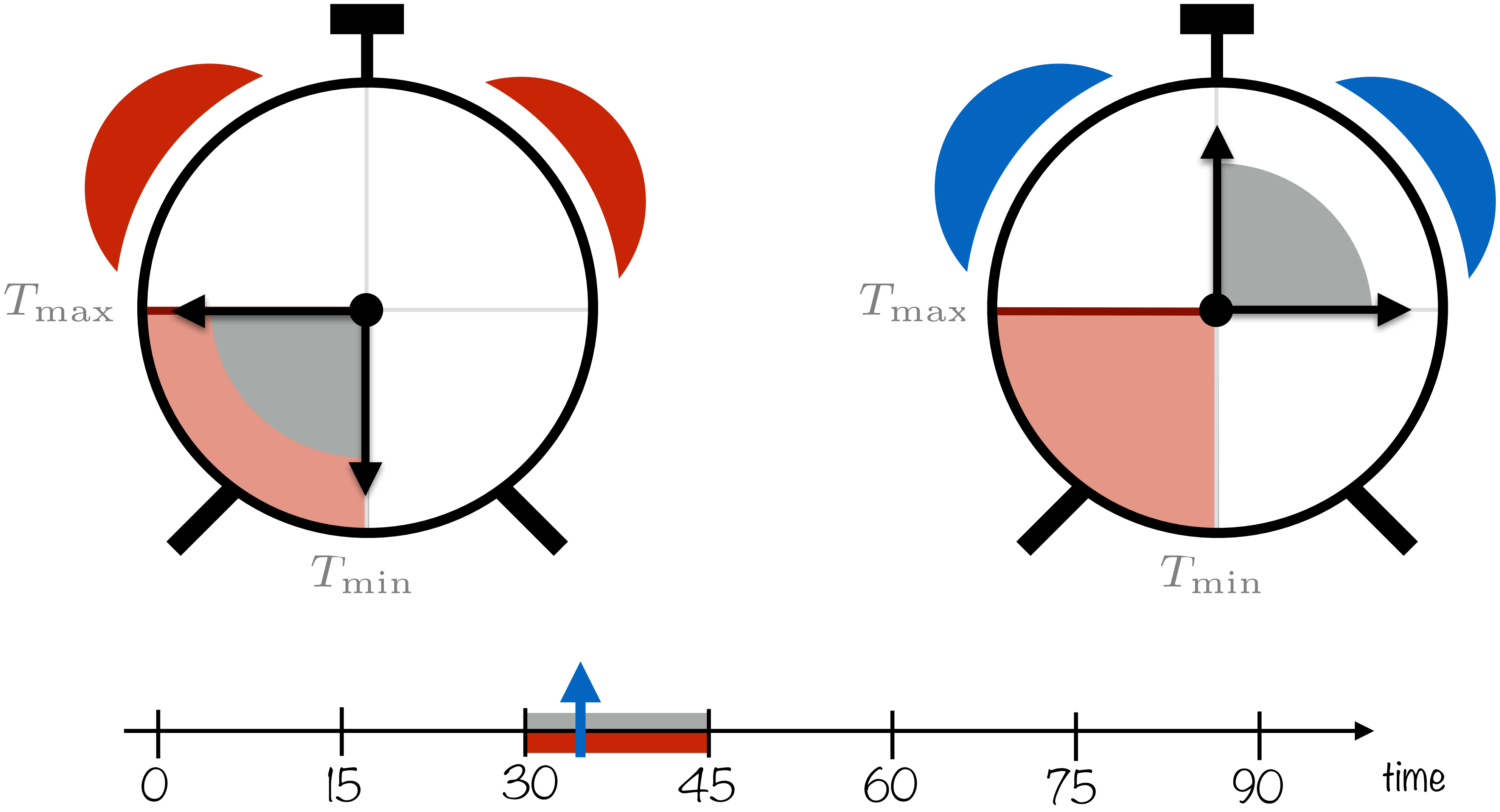


Symbolic simulation: capture multiple executions, using DBMs

Zones characterized by a set of possible choices

Symbolic Simulation

Example: a 2-node quasi-periodic architecture

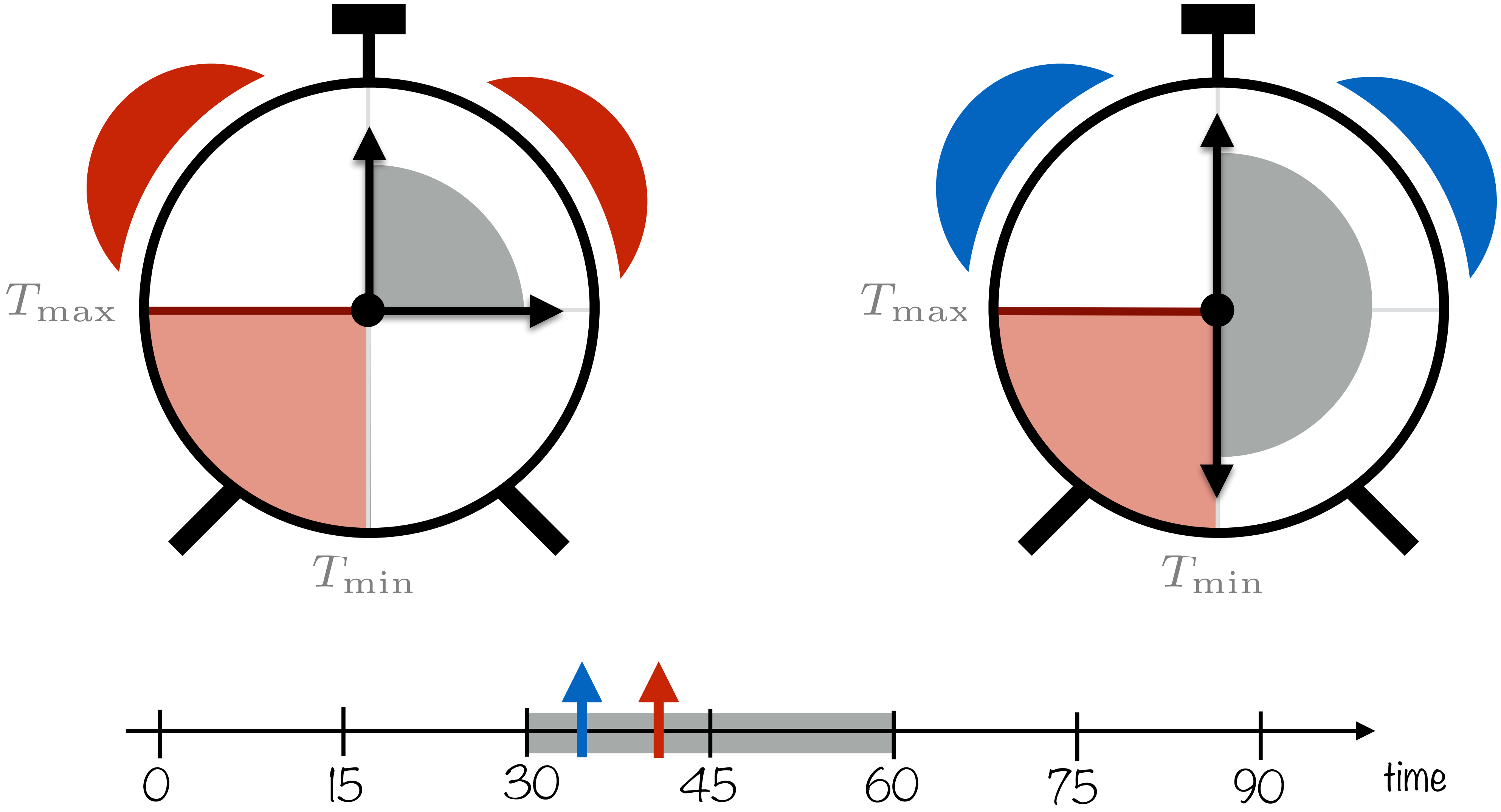


Symbolic simulation: capture multiple executions, using DBMs

Zones characterized by a set of possible choices

Symbolic Simulation

Example: a 2-node quasi-periodic architecture

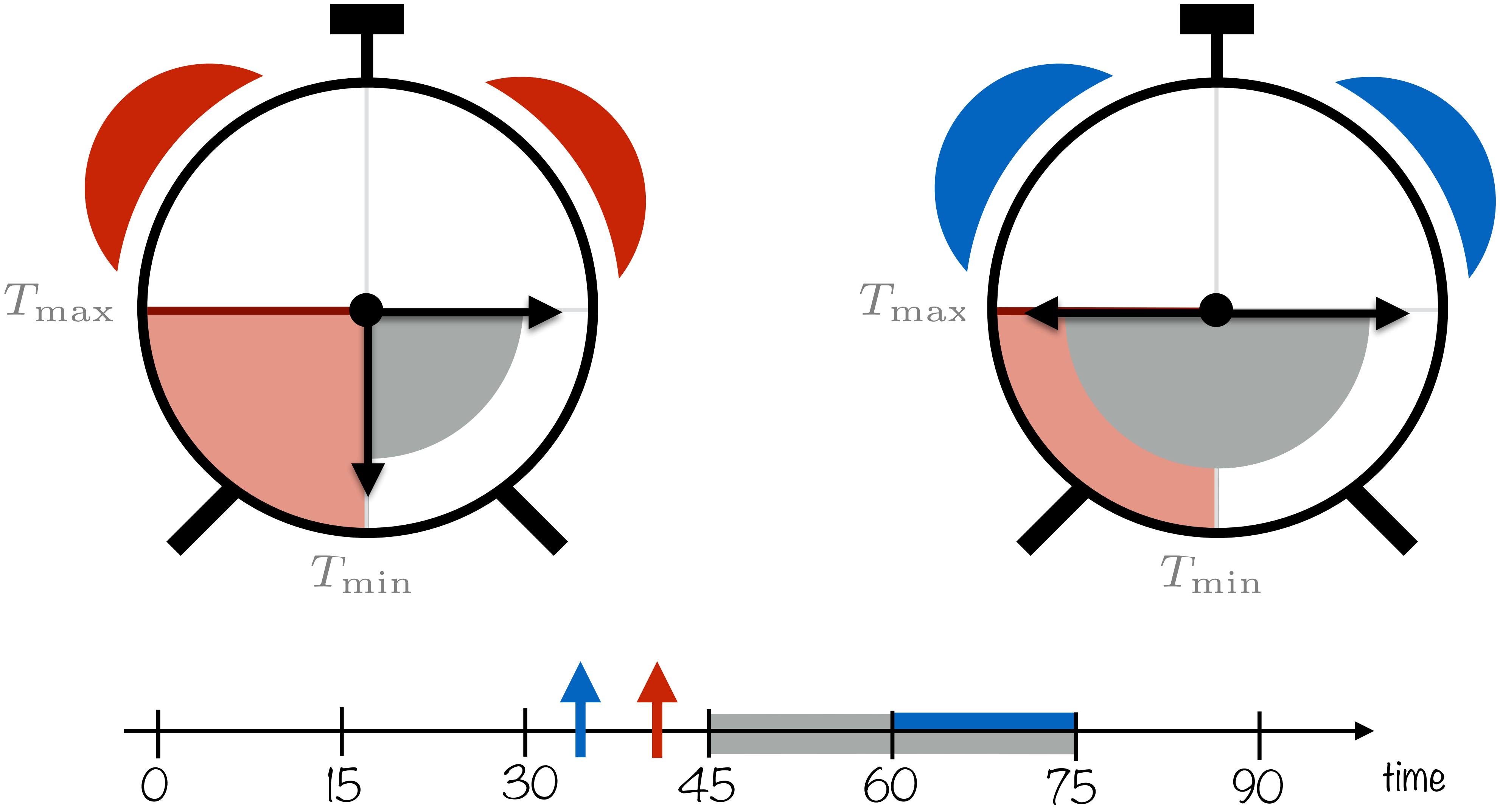


Symbolic simulation: capture multiple executions, using DBMs

Zones characterized by a set of possible choices

Symbolic Simulation

Example: a 2-node quasi-periodic architecture

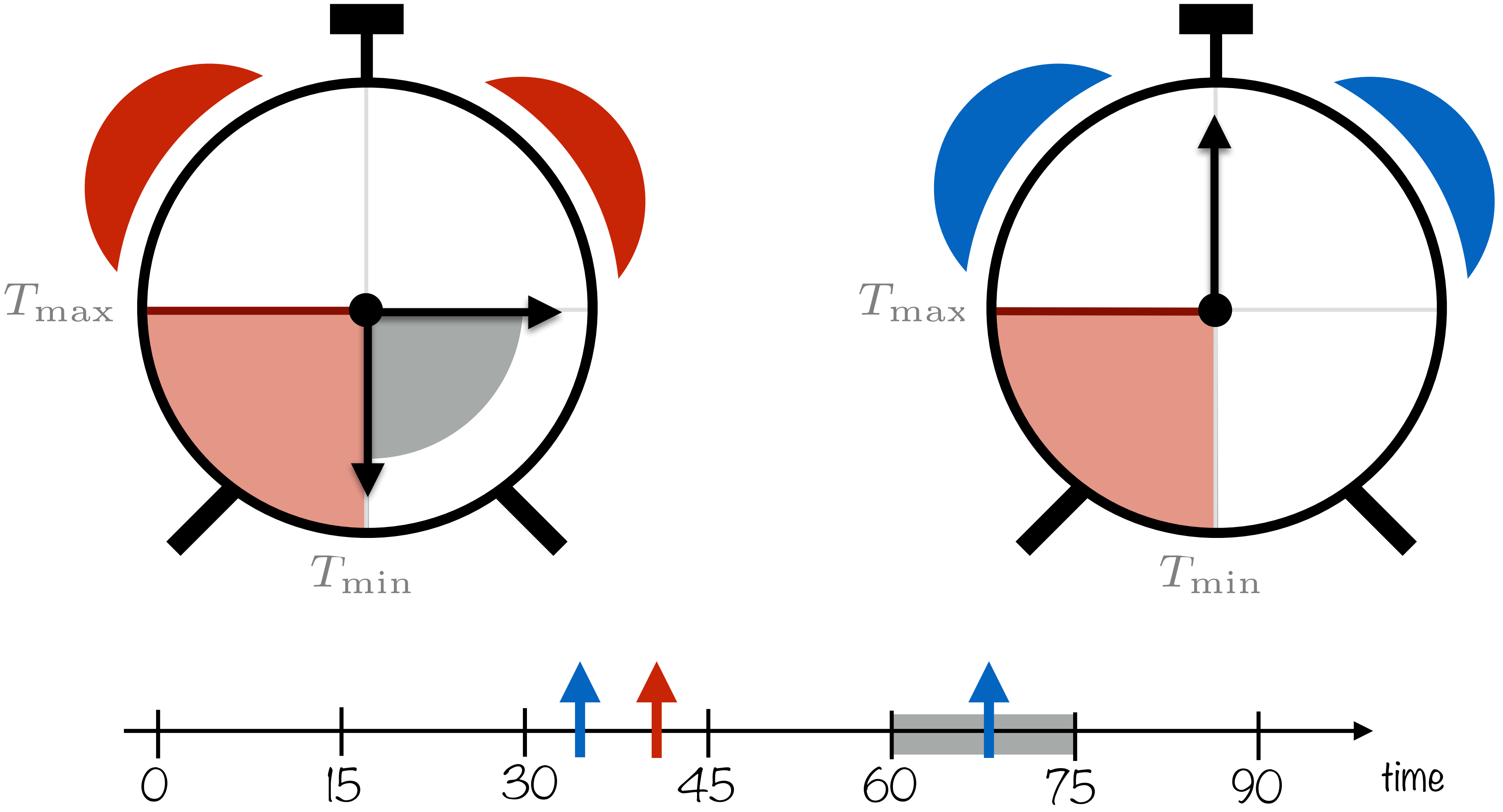


Symbolic simulation: capture multiple executions, using DBMs

Zones characterized by a set of possible choices

Symbolic Simulation

Example: a 2-node quasi-periodic architecture

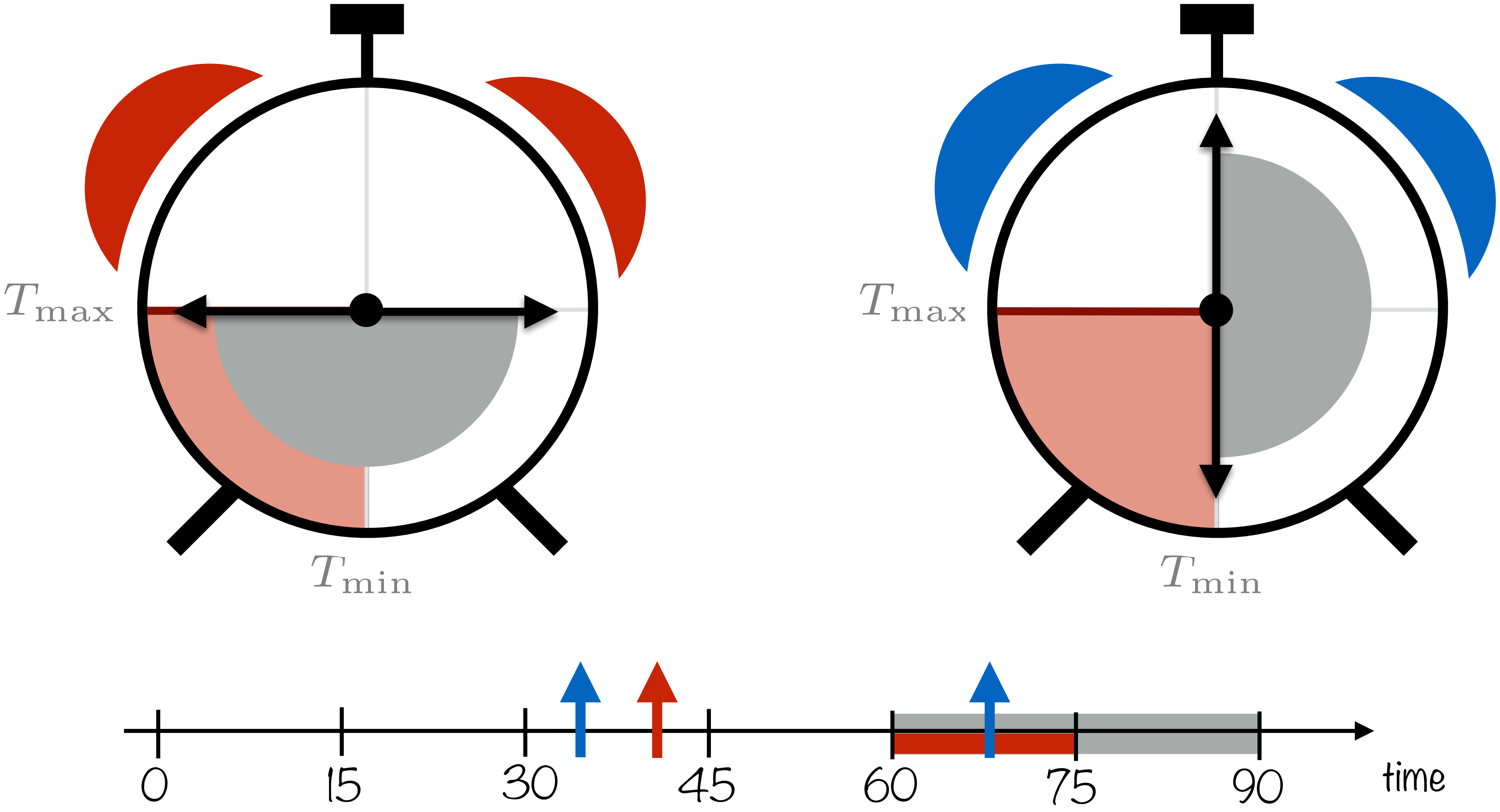


Symbolic simulation: capture multiple executions, using DBMs

Zones characterized by a set of possible choices

Symbolic Simulation

Example: a 2-node quasi-periodic architecture

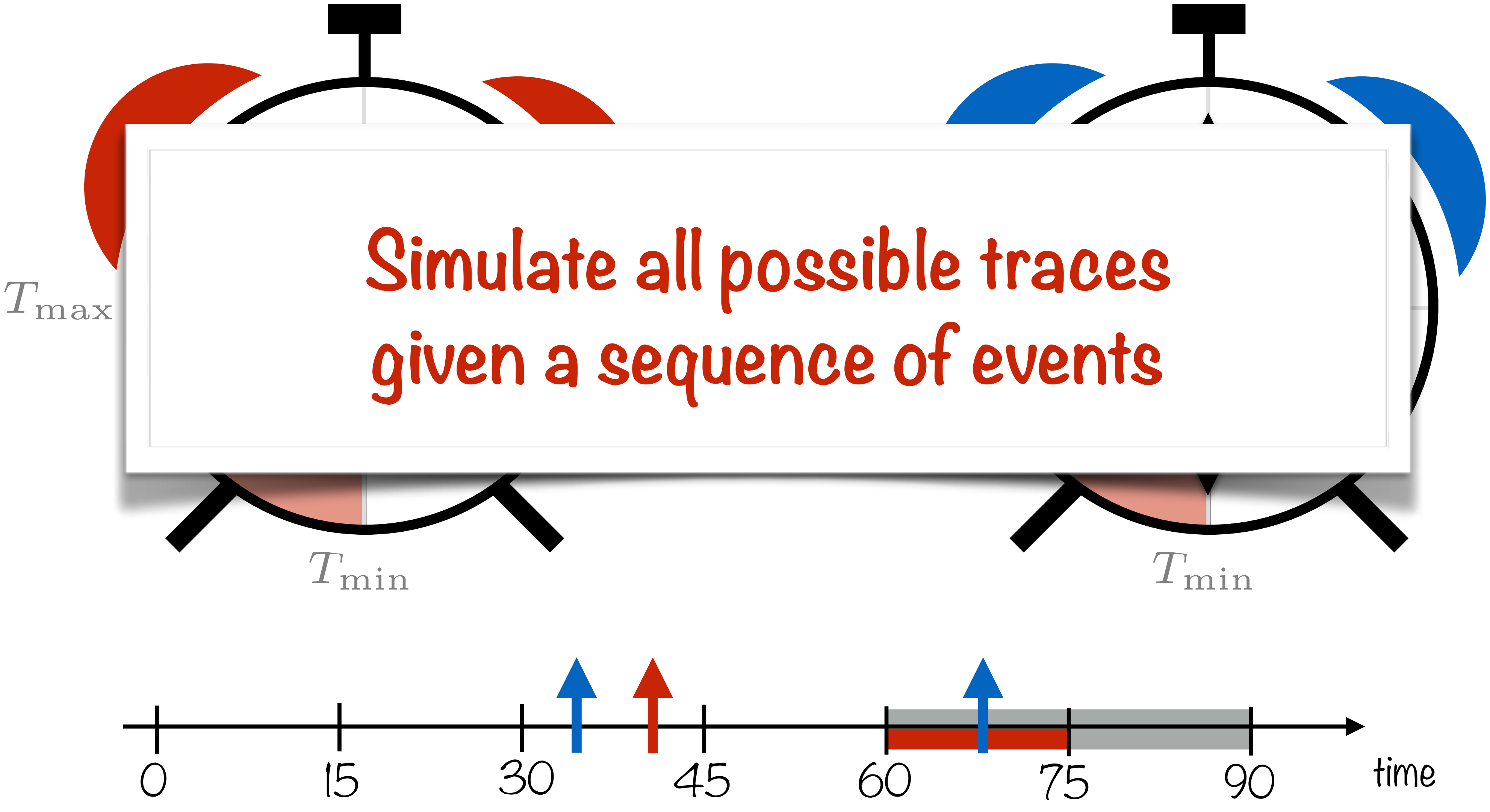


Symbolic simulation: capture multiple executions, using DBMs

Zones characterized by a set of possible choices

Symbolic Simulation

Example: a 2-node quasi-periodic architecture



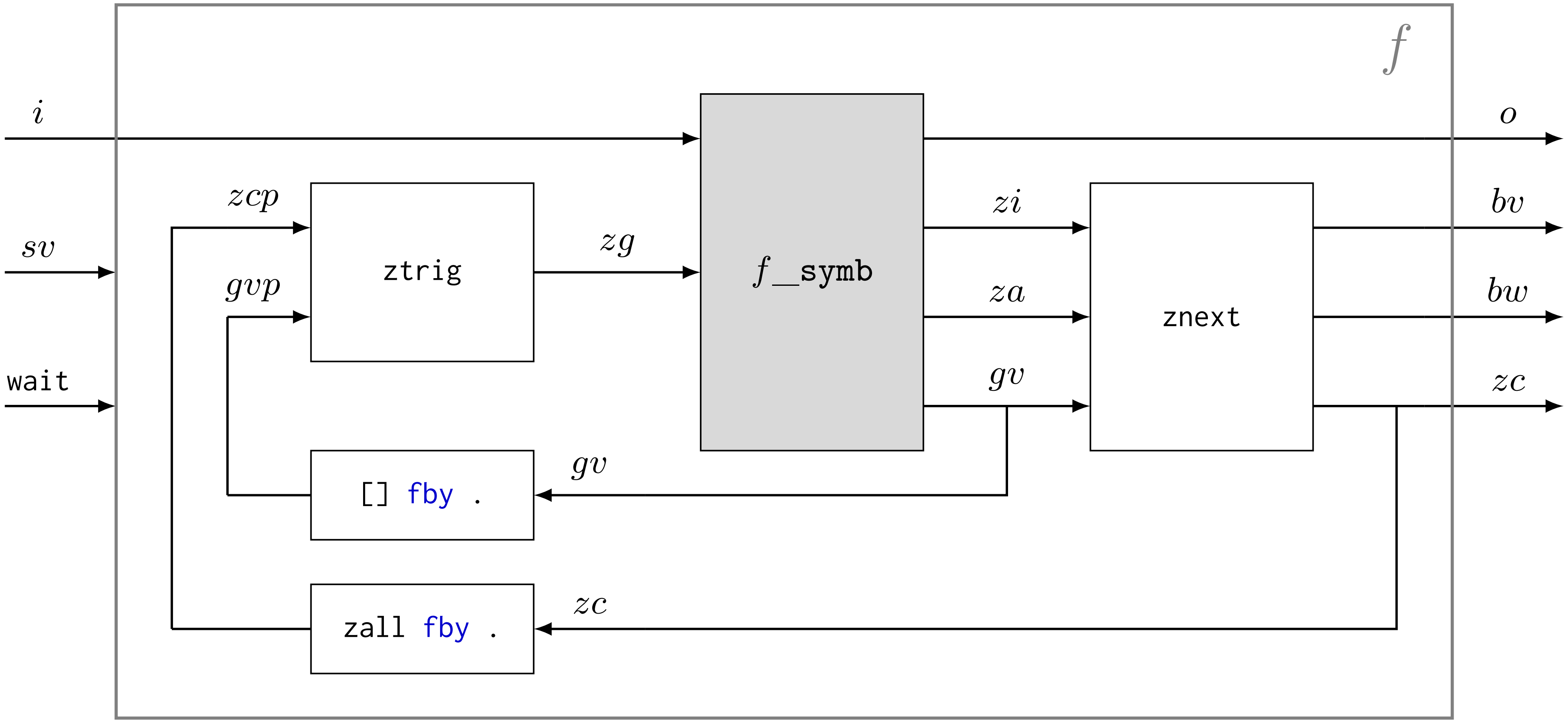
Simulate all possible traces given a sequence of events

Symbolic simulation: capture multiple executions, using DBMs

Zones characterized by a set of possible choices

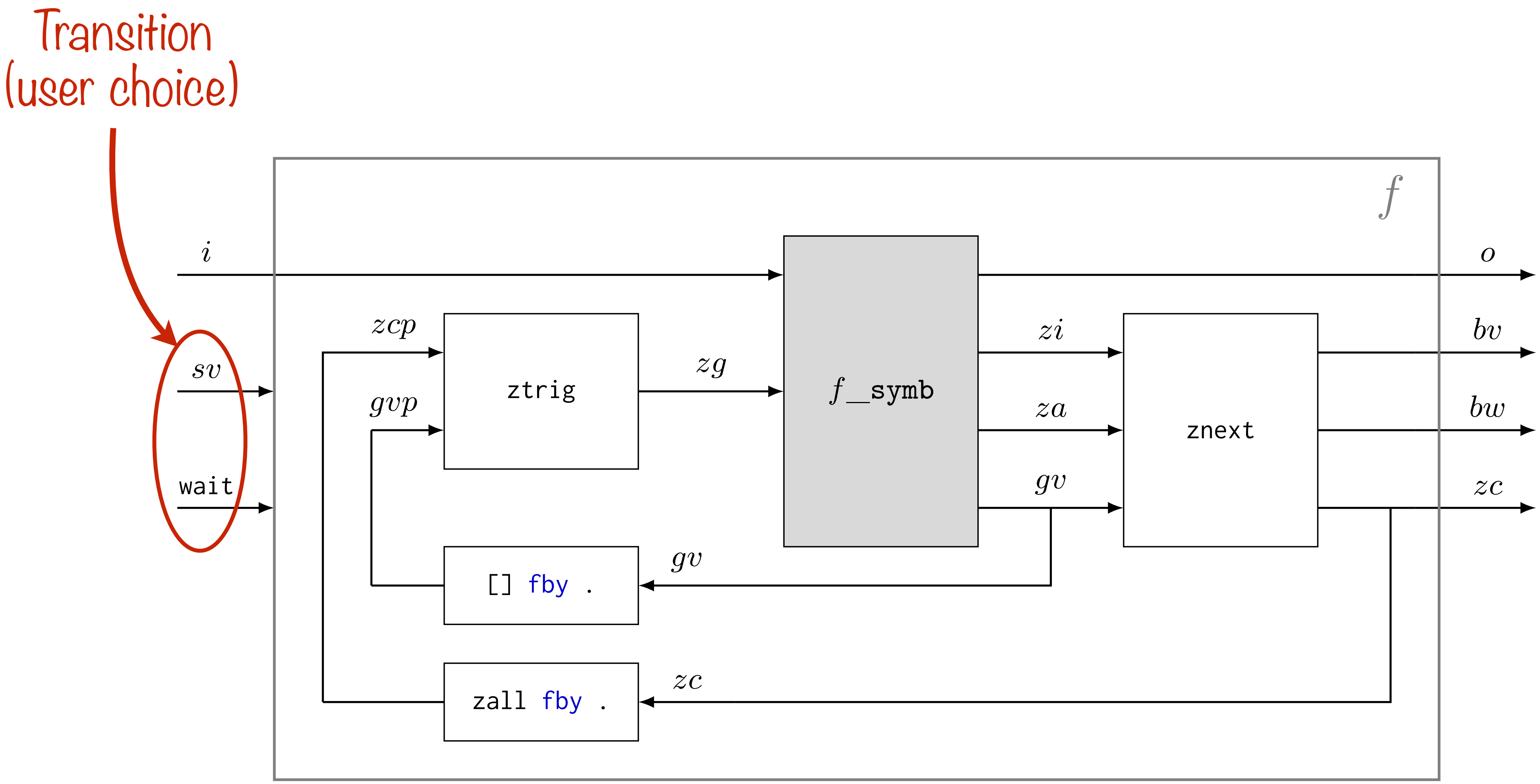
Symbolic Runtime

Compute the succession of zones



Symbolic Runtime

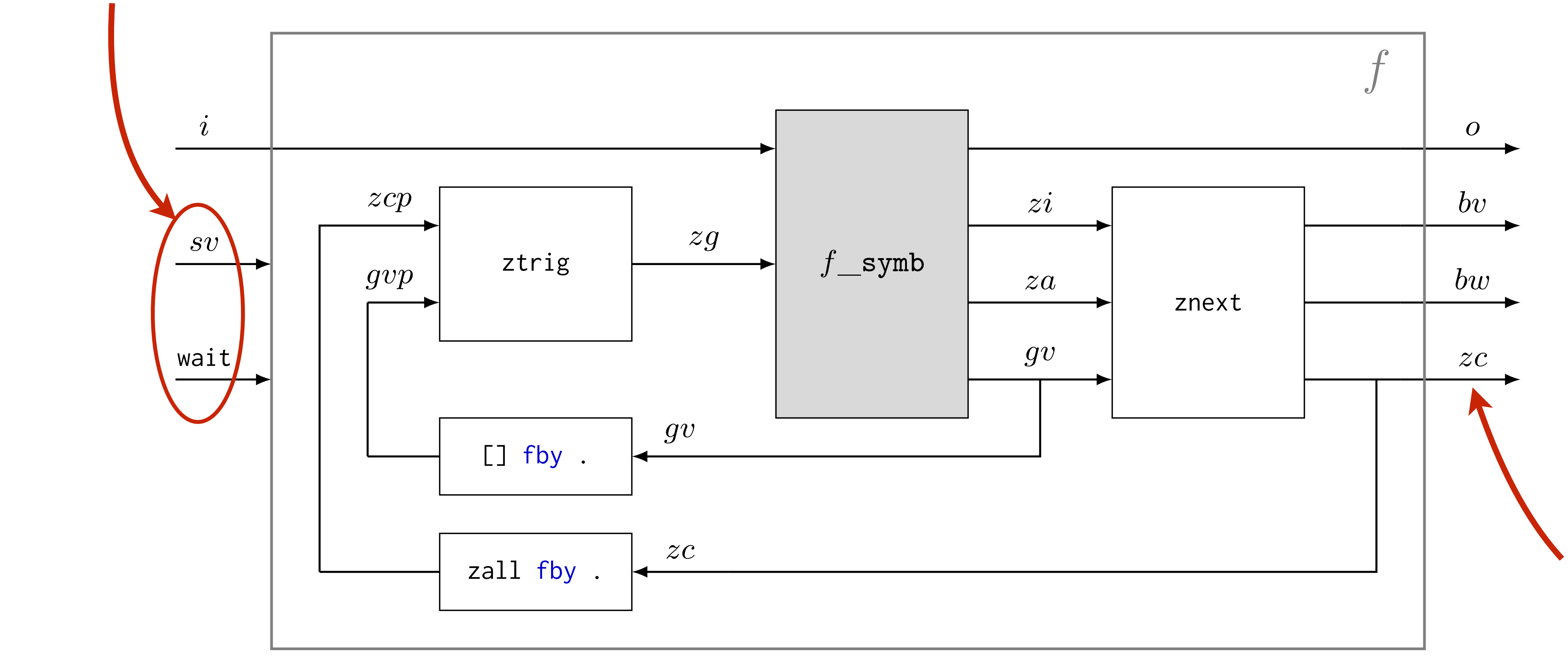
Compute the succession of zones



Symbolic Runtime

Compute the succession of zones

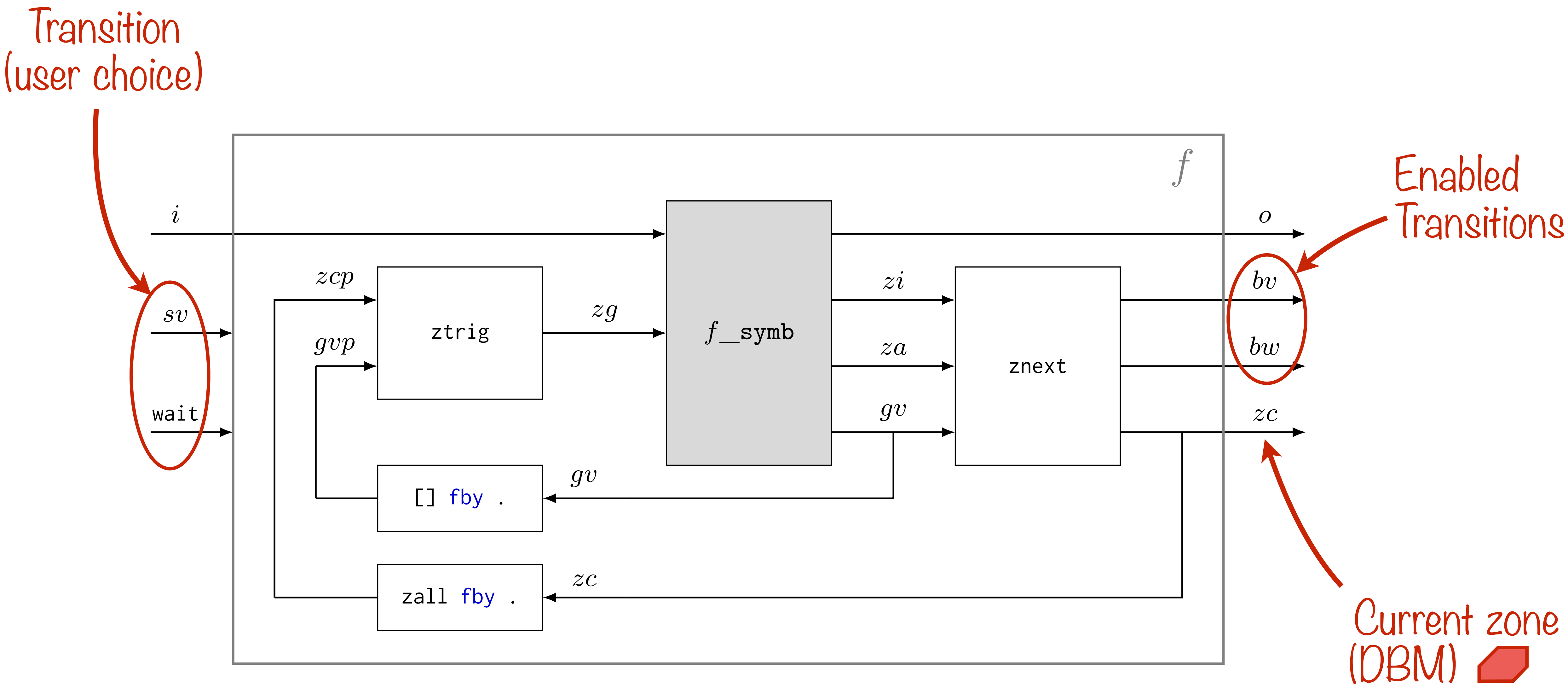
Transition
(user choice)



Current zone
(DBM) ▰

Symbolic Runtime

Compute the succession of zones



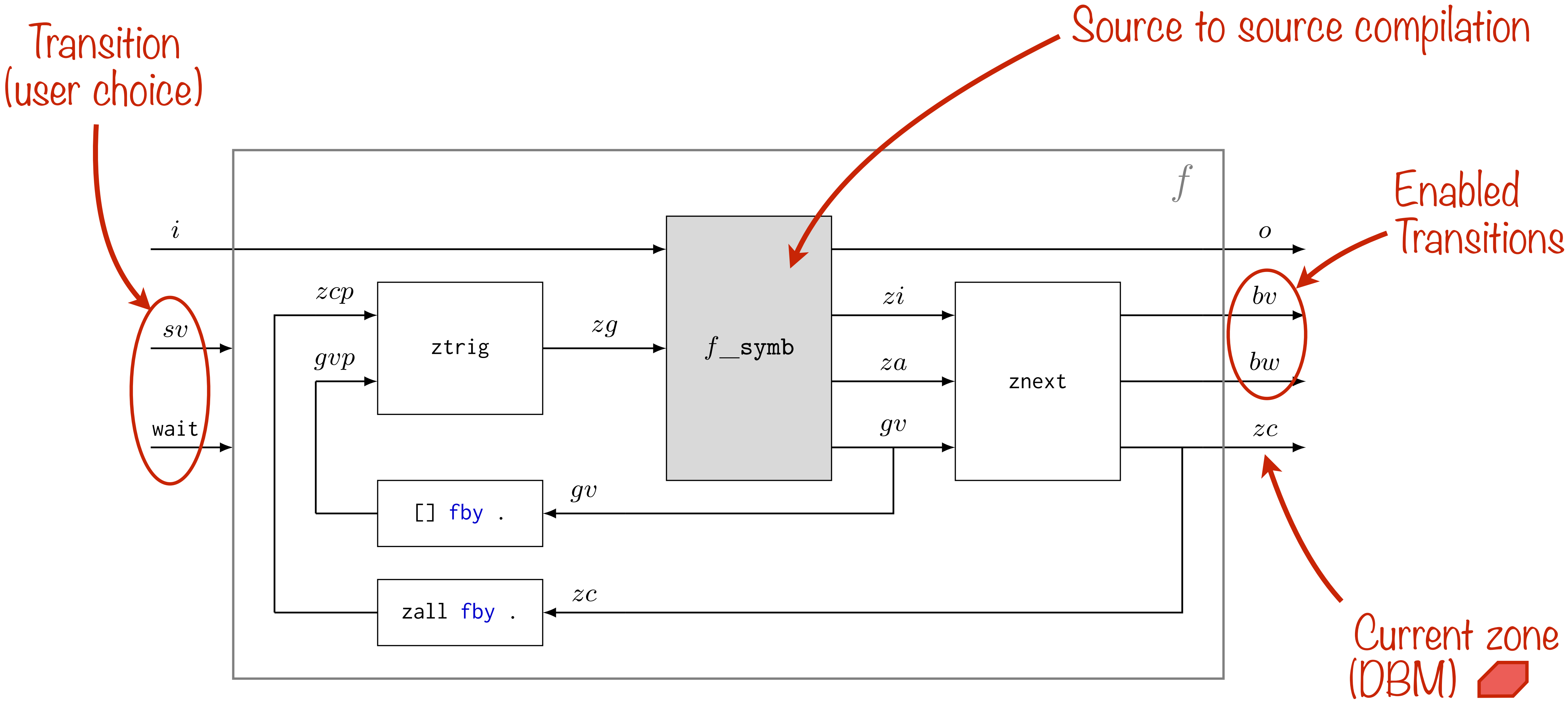
Transition (user choice)

Enabled Transitions

Current zone (DBM)

Symbolic Runtime

Compute the succession of zones



Transition (user choice)

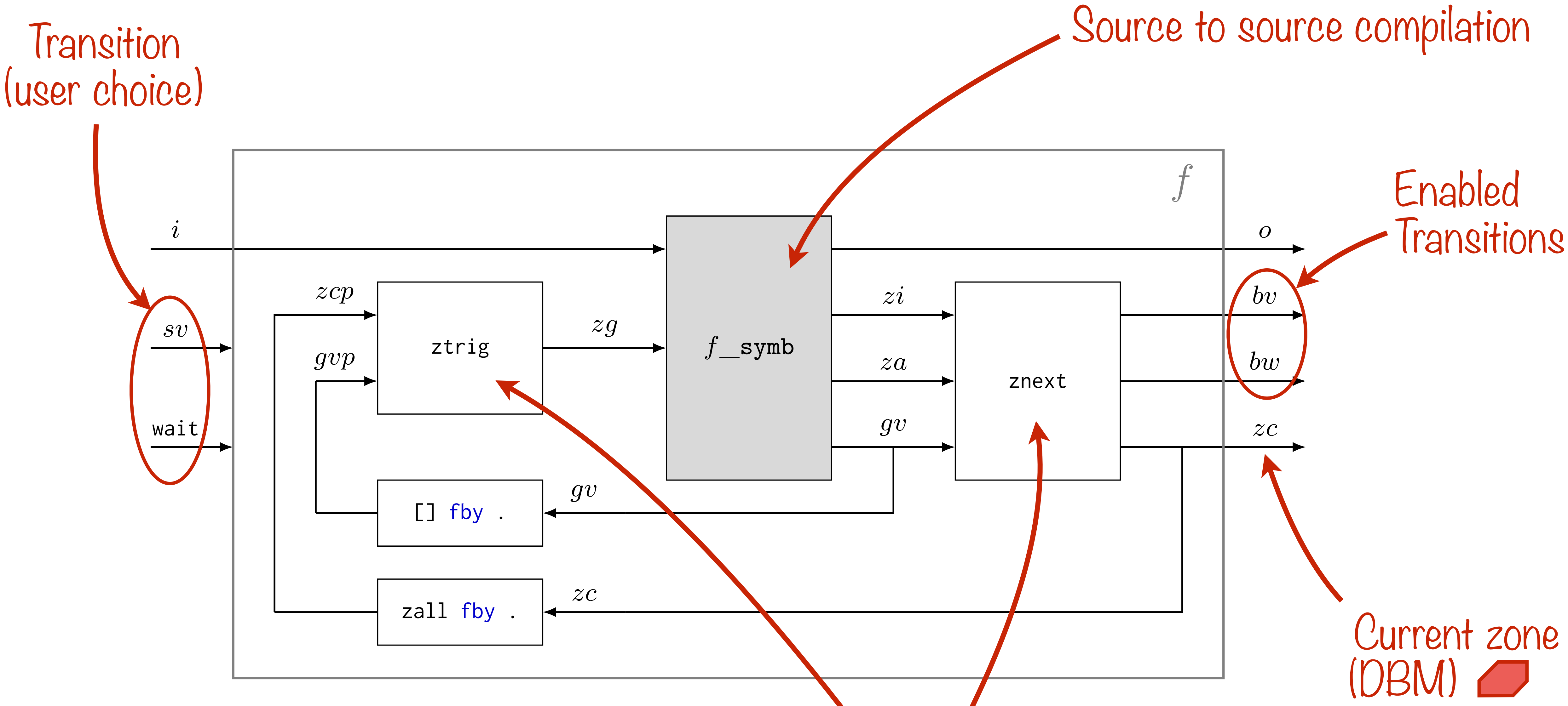
Source to source compilation

Enabled Transitions

Current zone (DBM)

Symbolic Runtime

Compute the succession of zones



Transition (user choice)

Source to source compilation

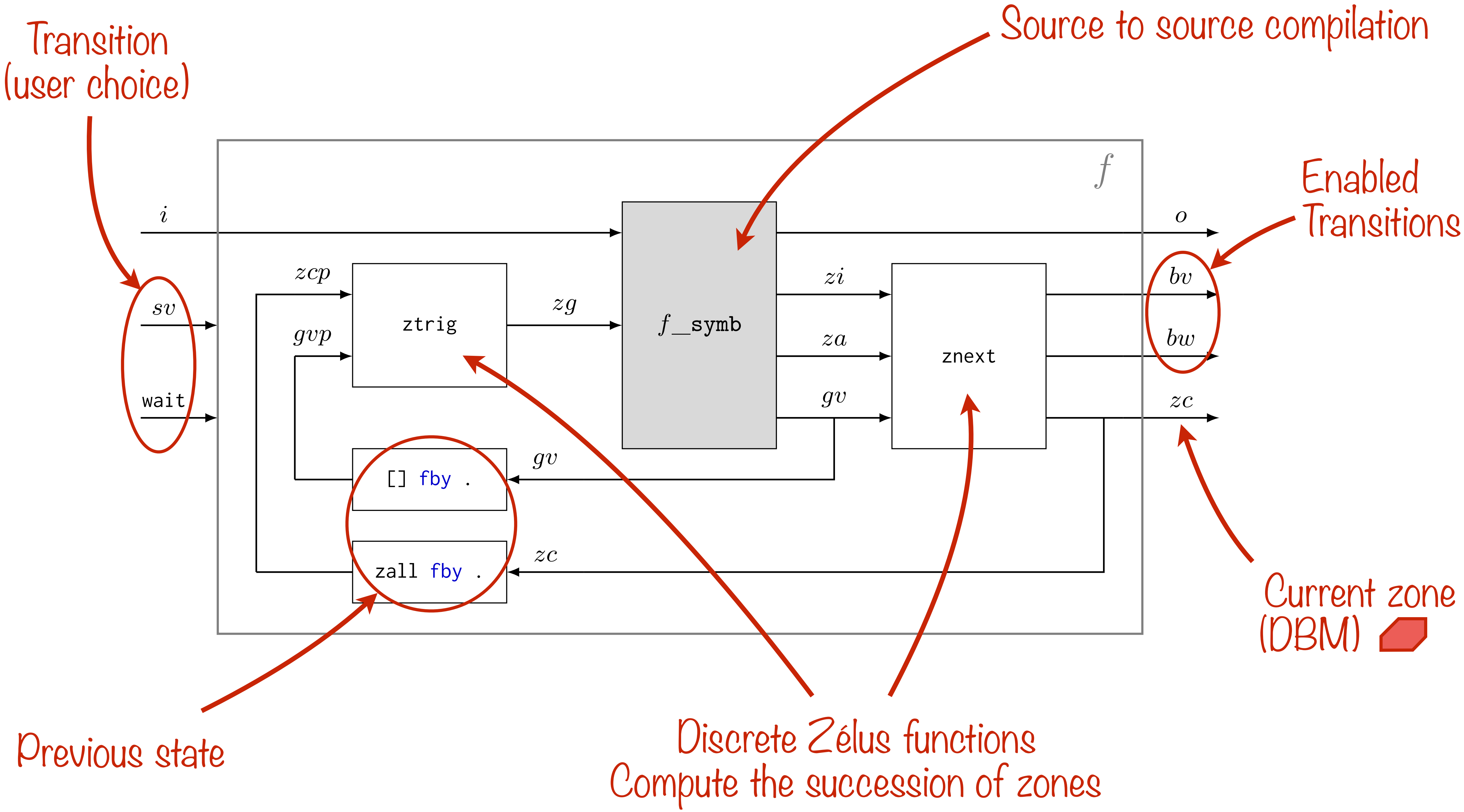
Enabled Transitions

Current zone (DBM)

Discrete Zélus functions
Compute the succession of zones

Symbolic Runtime

Compute the succession of zones



Source to Source Compilation

Continuous components are compiled into discrete function manipulating zones

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

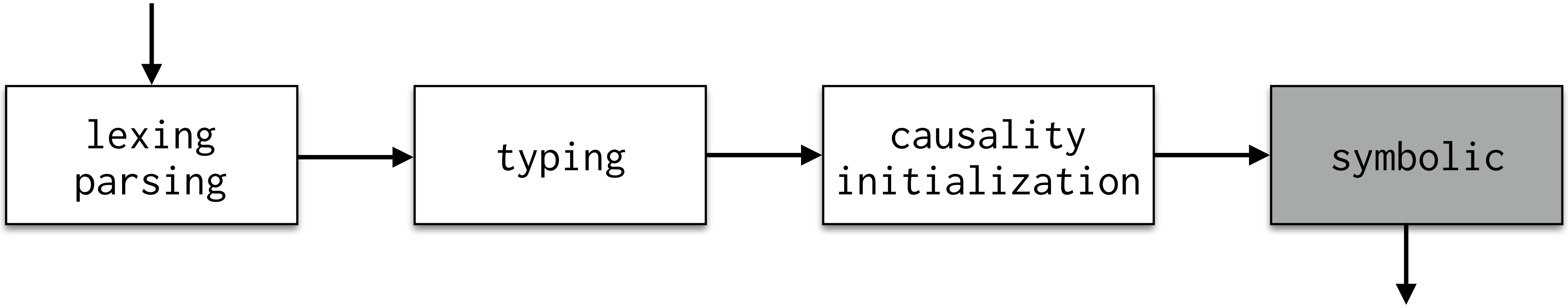
Continuous / Nondeterministic

Source to Source Compilation

Continuous components are compiled into discrete function manipulating zones

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

Continuous / Nondeterministic



lexing
parsing

typing

causality
initialization

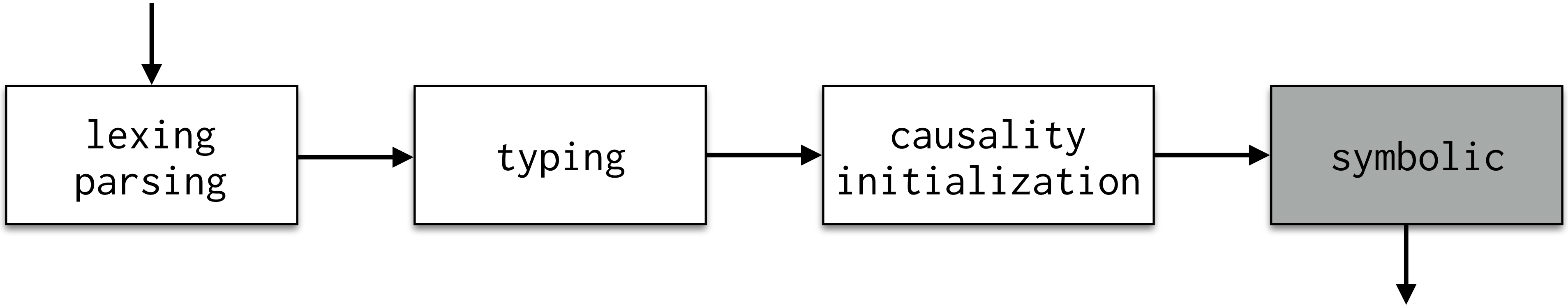
symbolic

Source to Source Compilation

Continuous components are compiled into discrete function manipulating zones

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

Continuous / Nondeterministic



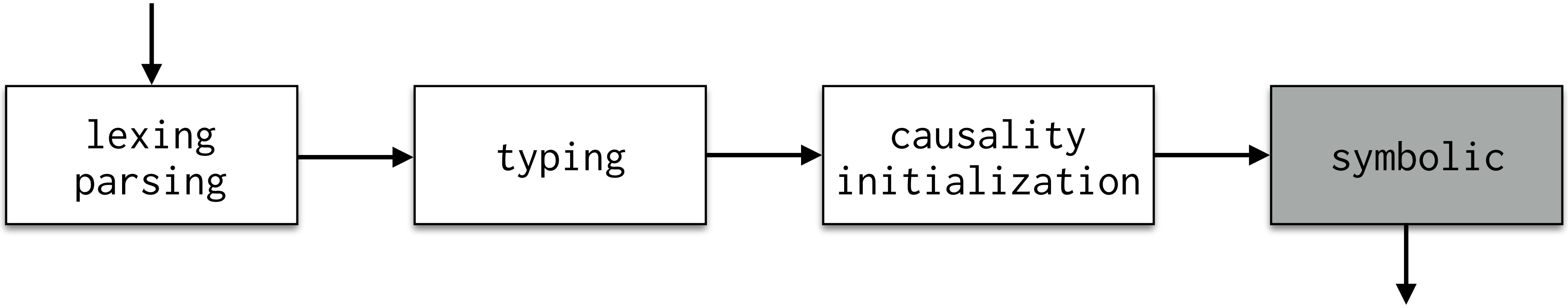
```
let node metro_symb(t, wait, c, zg, (t_min, t_max)) = c, zi, za, [zs] where
  rec zit = present (true fby false) → zreset(zg, t, 0)
              | c → zreset(zg, t, 0)
              else zg
  and zs = zmake({t ≥ t_min})
  and zb = zmake({t ≤ t_max})
  and za = zinterfold([zb])
  and zi = if wait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

Source to Source Compilation

Continuous components are compiled into discrete function manipulating zones

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

Continuous / Nondeterministic



```
let node metro_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
```

Discrete / Deterministic

Manipulate zones

Transitions controlled by the user

Prototype Implementation

```
let hybrid metro(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 archi(t_min, t_max) = c1, c2 where
  rec c1 = metro(t_min, t_max)
  and c2 = metro(t_min, t_max)
```

Prototype Implementation

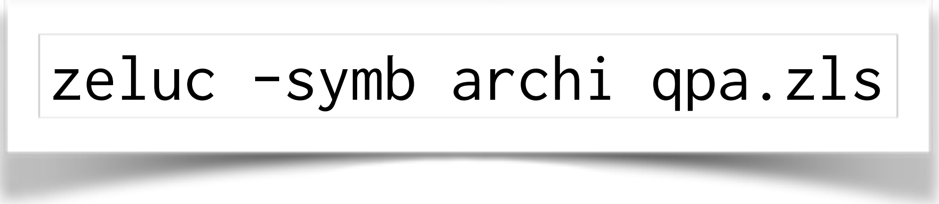
```
let hybrid metro(t_min, t_max) = c where
  rec timer t init 0 reset c() → 0
  and emit c when {t ≥ t_min}
  and always {t ≤ t_max}

let hybrid archi(t_min, t_max) = c1, c2 where
  rec c1 = metro(t_min, t_max)
  and c2 = metro(t_min, t_max)
```

```
zeluc -symb archi qpa.zls
```

```
let node metro_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

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

(*** Runtime ***)
let node archi(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 = archi_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
  and gvp = [] fby gv
```

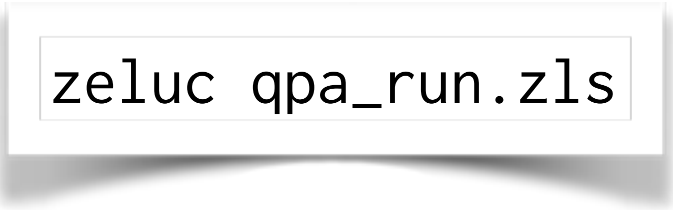
Prototype Implementation

```
let hybrid metro(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 archi(t_min, t_max) = c1, c2 where
  rec c1 = metro(t_min, t_max)
  and c2 = metro(t_min, t_max)
```



```
zeluc -symb archi qpa.zls
```

```
let node metro_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

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

(*** Runtime ***)
let node archi(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 = archi_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
  and gvp = [] fby gv
```



```
zeluc qpa_run.zls
```

Conclusion

Verification

Verifying safety properties of quasi-periodic systems

The Quasi-Synchronous Abstraction

Implementation

Deploying code on quasi-periodic architectures

Loosely Time-Triggered Architectures

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

Conclusion

Verification

Verifying safety properties of quasi-periodic systems

The Quasi-Synchronous Abstraction

Implementation

Deploying code on quasi-periodic architectures

Loosely Time-Triggered Architectures

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

Abstraction is not sound in general

Give exact conditions of application

Generalization to multirate systems

Conclusion

Verification

Verifying safety properties of quasi-periodic systems

The Quasi-Synchronous Abstraction

Abstraction is not sound in general

Give exact conditions of application

Generalization to multirate systems

Implementation

Deploying code on quasi-periodic architectures

Loosely Time-Triggered Architectures

Unified synchronous framework

Executable specifications

Correctness proofs

Optimizations and comparisons

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

Conclusion

Verification

Verifying safety properties of quasi-periodic systems

The Quasi-Synchronous Abstraction

Abstraction is not sound in general

Give exact conditions of application

Generalization to multirate systems

Implementation

Deploying code on quasi-periodic architectures

Loosely Time-Triggered Architectures

Unified synchronous framework

Executable specifications

Correctness proofs

Optimizations and comparisons

Simulation

Simulating the possible behaviors of quasi-periodic systems

Symbolic Simulation

Zélus extended with timed nondeterminism

Symbolic simulation

Modular source-to-source compilation

Prototype implementation

Open Questions

Real-time requirements
LTTAs preserve the semantics at the cost of additional latency
Not acceptable for all applications (emergency button)
What is the impact of these delays on the application?

Open Questions

Real-time requirements
LTTAs preserve the semantics at the cost of additional latency
Not acceptable for all applications (emergency button)
What is the impact of these delays on the application?

Characterizing robust applications
Some applications are already robust to sampling artifacts (3-voters)
How to check this property on a given application?
What is the impact of the sampling artifacts on the semantics?

Open Questions

Real-time requirements
LTTAs preserve the semantics at the cost of additional latency
Not acceptable for all applications (emergency button)
What is the impact of these delays on the application?

Characterizing robust applications
Some applications are already robust to sampling artifacts (3-voters)
How to check this property on a given application?
What is the impact of the sampling artifacts on the semantics?

Zélus in a proof assistant
Formalization of a the semantics mixing discrete and continuous time
Prove properties involving real-time specifications (Time-Based LTTA)

Open Questions

Real-time requirements
LTTAs preserve the semantics at the cost of additional latency
Not acceptable for all applications (emergency button)
What is the impact of these delays on the application?

Characterizing robust applications
Some applications are already robust to sampling artifacts (3-voters)
How to check this property on a given application?
What is the impact of the sampling artifacts on the semantics?

Zélus in a proof assistant
Formalization of a the semantics mixing discrete and continuous time
Prove properties involving real-time specifications (Time-Based LTTA)

Model checking
Explore all possible simulation choices (symbolic simulation)
Reuse existing technique for model checking timed systems (Uppaal)
Model check the generated code with Kind2 and Lesar

[EMSOFT'13] **A Synchronous Embedding of Antescofo, a Domain-Specific Language for Interactive Mixed Music**, with Florent Jacquemard, Louis Mandel, and Marc Pouzet
International Conference on Embedded Software (EMSOFT) 2013

[FARM'13] **Programming Mixed-Music in ReactiveML**,
with Louis Mandel and Marc Pouzet
ICFP Workshop on Functional Art, Music, Modeling and Design (FARM) 2013

[EMSOFT'15] **Loosely Time-Triggered Architectures: Improvements and Comparisons,**
with Timothy Bourke and Albert Benveniste
International Conference on Embedded Software (EMSOFT) 2015

[TECS'16] **Loosely Time-Triggered Architectures: Improvements and Comparisons,**
with Timothy Bourke and Albert Benveniste
ACM Transaction on Embedded Computing Systems (TECS) 2016

[FMCAD'16] **Soundness of the Quasi-Synchronous Abstraction**,
with Timothy Bourke and Marc Pouzet
International Conference on Formal Methods in Computer-Aided Design (FMCAD) 2016

[JFLA'17] **CloudLens, un langage de script pour l'analyse de données semi-structurées**
with Louis Mandel, Olivier Tardieu, and Mandana Vaziri
Journées Francophone des Langages Applicatifs (JFLA) 2017

[Submitted] **CloudLens, a scripting language for semi-structured data**
with Louis Mandel, Olivier Tardieu, and Mandana Vaziri