TECHNISCHE UNIVERSITÄT DRESDEN

DRESDEN concept

Lingua Franca a Language to Coordinate the RIOT

Tassilo Tanneberger, TU Dresden and UC Berkeley

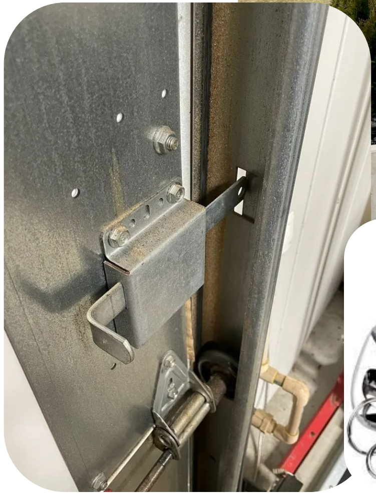
September 6, 2024

Today’s topic ;).

An illustrative example: Smartifying a garage

- Motor for opening and closing.
- Lock to secure against thieves.
- Small stepper motor for the lock.
- Small microcontroller
  to control it all.

- Now Firmware!



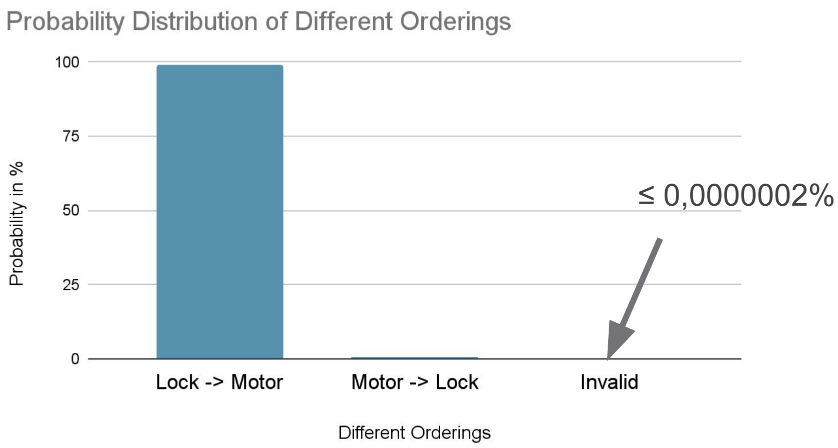
OS Synchronisation: Do you spot the problem?

```
class GarageDoor {
private:
      int state_ = 0;
public:
      void lock() {
            state_ = 0;
            // Lock Door ...
      }

      void motor() {
            state_ = 1;
            // Actuate Motor ...
      }
};
```

OS Synchronisation: Do you spot the problem?

```
class GarageDoor {
private:
      int state_ = 0;
public:
      void lock() {
            state_ = 0;
            // Lock Door ...
      }

      void motor() {
            state_ = 1;
            // Actuate Motor ...
      }
};
```

- ❌ **Potentially invalid state (when preemptable)**
- ❌ **No ordering is enforced.**

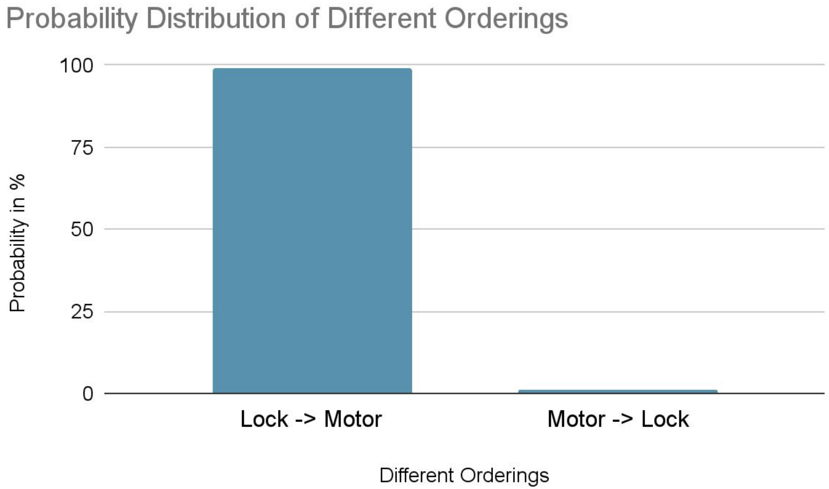


OS Synchronisation - Step (1): Mutual exclusion

```cpp
class GarageDoor {
private:
    std::mutex mutex_;
    int state_ = 0;
public:
    void lock() {
        std::lock_guard<std::mutex> m(mutex_);
        state_ = 0;
        // Lock Door ...
    }

    void motor() {
        std::lock_guard<std::mutex> m(mutex_);
        state_ = 1;
        // Actuate Motor ...
    }
};
```

✅ **Fixed Invalid State**

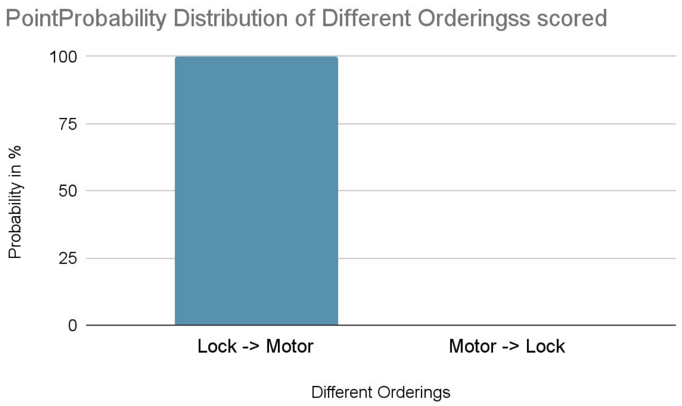
❌ **No Ordering is enforced.**



OS Synchronisation - Step (2): Enforced ordering

```cpp
class GarageDoor {
private:
    std::mutex mutex_;
    std::vector<Event> events_;
    int state_ = 1;
public:
    void lock() {
        std::lock_guard<std::mutex> m(mutex_);
        events_.push_back(Event::Lock);
    }

    void motor() {
        std::lock_guard<std::mutex> m(mutex_);
        events_.push_back(Event::Motor);
    }

    auto process() -> int {
        std::lock_guard<std::mutex> m(mutex_);

        if (std::find(events_.begin(), events_.end(), Event::Lock) != std::end(events_)) {
            state_ = 0;
            // Lock Door ...
        }

        if (std::find(events_.begin(), events_.end(), Event::Motor) != std::end(events_)) {
            if (state_ == 0) {
                // Problem ... door locked but we want to move it.
            } else {
                state_ = 1;
                // Actuate Motor ...
            }
        }

        return state_;
    }
};
```

- ✅ **Fixed Invalid State**
- ✅ **Lock is always processed before motor.**
- ❌ **Complicated and Error-Prone**



What is this talk about?

1. Highlight the problems and prevalence of (unwanted) non-determinism.

2. Offering one solution: Lingua Franca.

3. Present the ongoing work to leverage the RIOT OS ecosystem.

Let’s consider a common IoT deployment

- **AUTOSAR, ROS, ROS2, … and many more**
- **Very tedious to enforce order, e.g., ROS message filters, bundling messages, extra meta-information**

Still not convinced that order matters?

1.) **Disarm**
2.) **Open**
⇒ **Door opens regularly**

Still not convinced that order matters? It does!

1.) Disarm
2.) Open
⇒ Door opens regularly

1.) Open
2.) Disarm
⇒ Emergency slides deploy

So what are the problems?

So what are the problems?

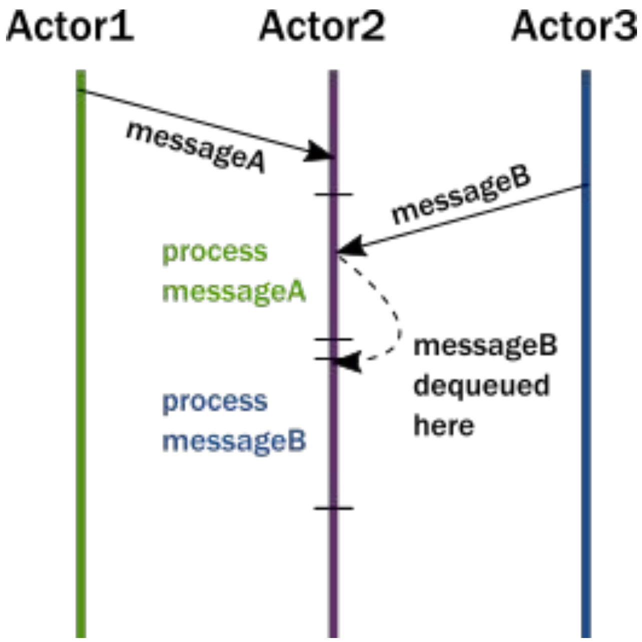
When do events count as simultaneously? (Timed)
⇒ What is the ordering of events in time? When are events synchronous?

How to ensure deterministic execution? (Determinism)
⇒ How to ensure the program executes deterministically.

How to make the system interactive? (Reactive)
⇒ How to make sure the system responds in time.

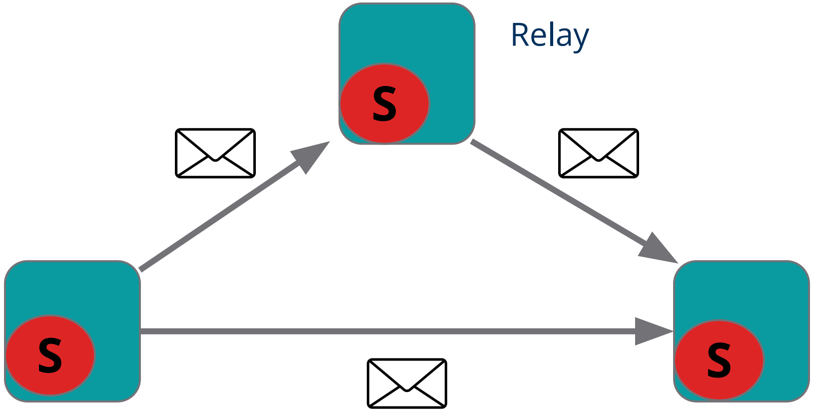
How to keep keep the overhead in check? (Scalable)
⇒ Larger systems should be still performant.

Option 1: Actors



© Akka Documentation

- Reactive & Scalable
- Ordering is still undefined
(not deterministic, not timed)



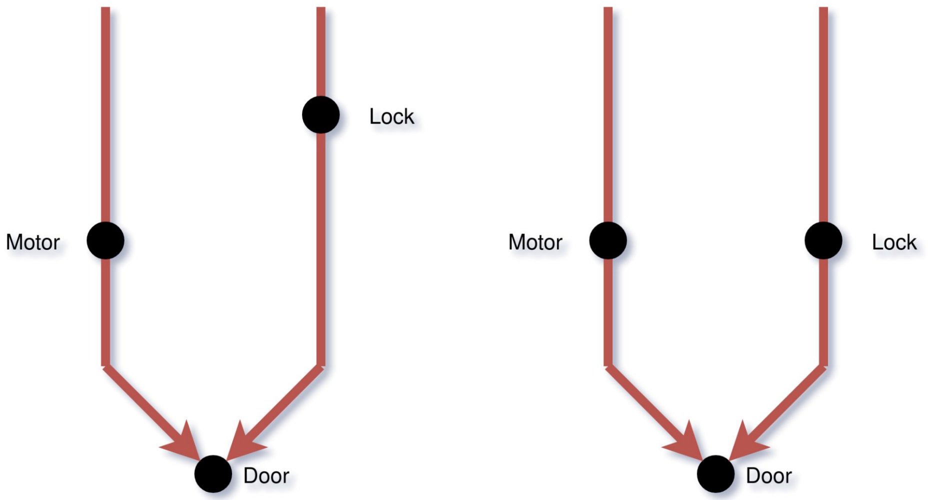
Option 2: Synchronous Languages

```
blink led =
    loop
        after ms 50,
            led <- not (deref led)
        wait led
```

(Esterel code that toggles the led signal every 50ms)

- ✅ **Timed: Clear semantics when (logically) events happen.**
- ✅ **Deterministic execution.**
- ❌ **Not scalable.**

Synchronous Languages & HW Synthesis



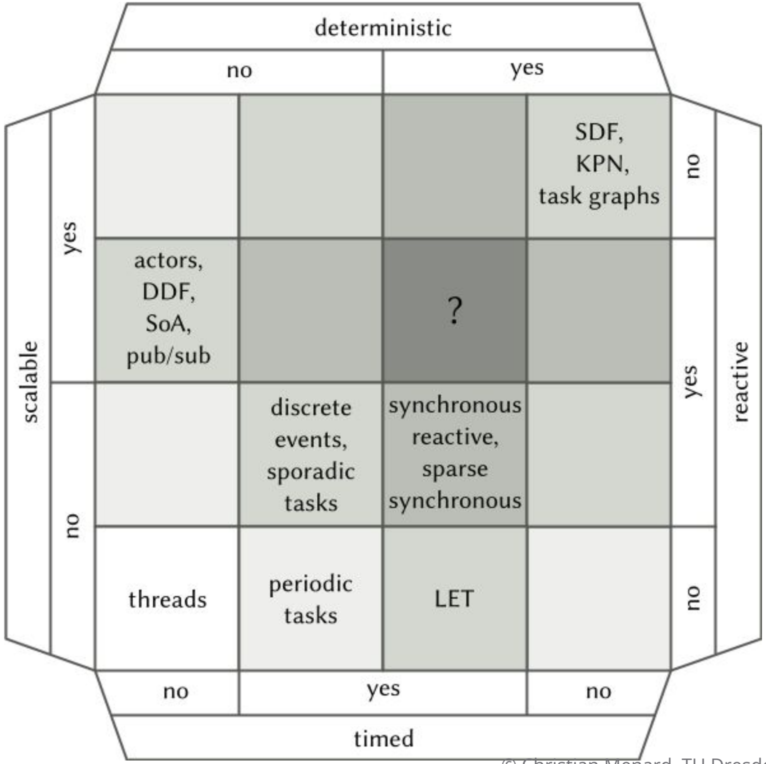
Assumptions

1.) Computation happens instantaneously.
2.) Transmission of Signals is instantaneous.
3.) Logical Ticks.

Present in

- Very common assumptions in hardware description languages such as Verilog and VHDL
- Synchronous Programming Languages, e.g., Esterel, Lustre, SIGNAL

Concurrency Concepts



Is there a combination?

LINGUA FRANCA

Berkeley
UNIVERSITY OF CALIFORNIA

TECHNISCHE UNIVERSITÄT DRESDEN

NTNU

ASU
ARIZONA STATE UNIVERSITY

C A U

xronos

TECHNISCHE UNIVERSITÄT DRESDEN

RIOT Summit
September 5 – 7, 2024

"Lingua Franca is a polyglot, declarative, coordination language for real-time, concurrent (and distributed) systems."

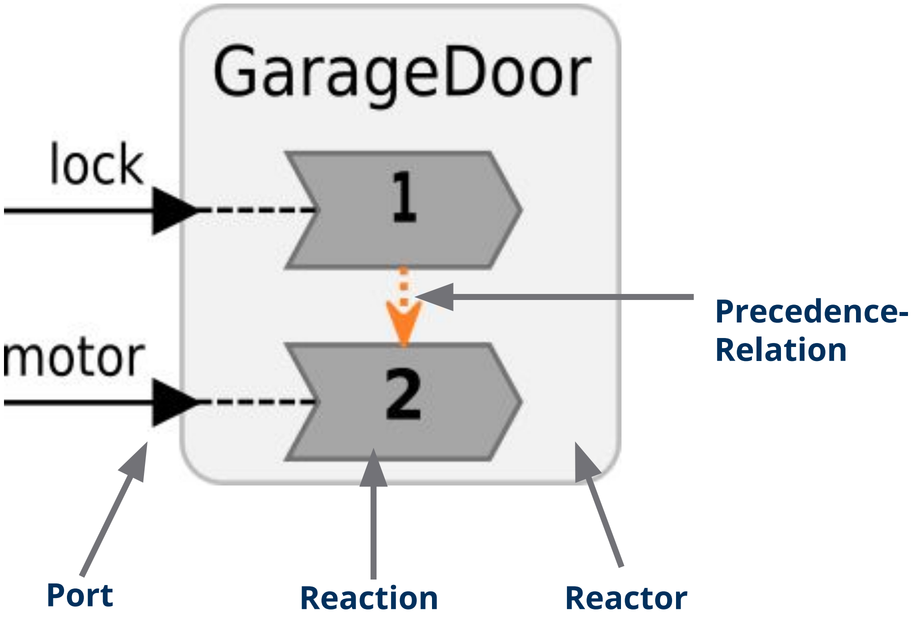
https://lf-lang.org

Marten Lohstroh

Christian Menard

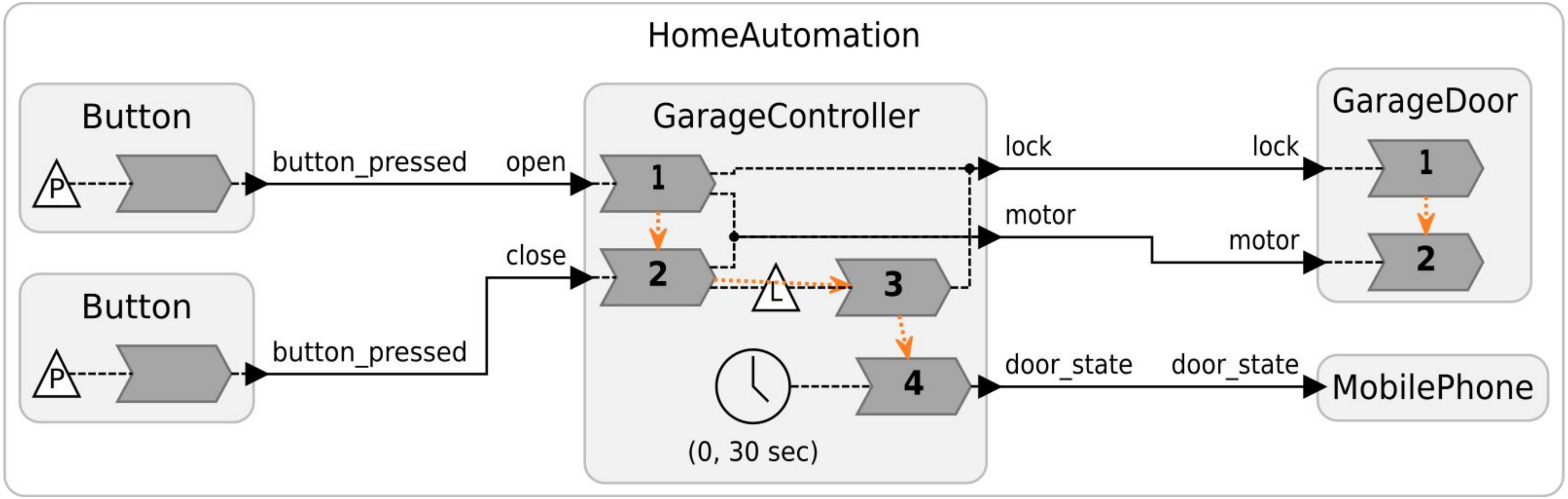
Edward A. Lee

Language Concepts

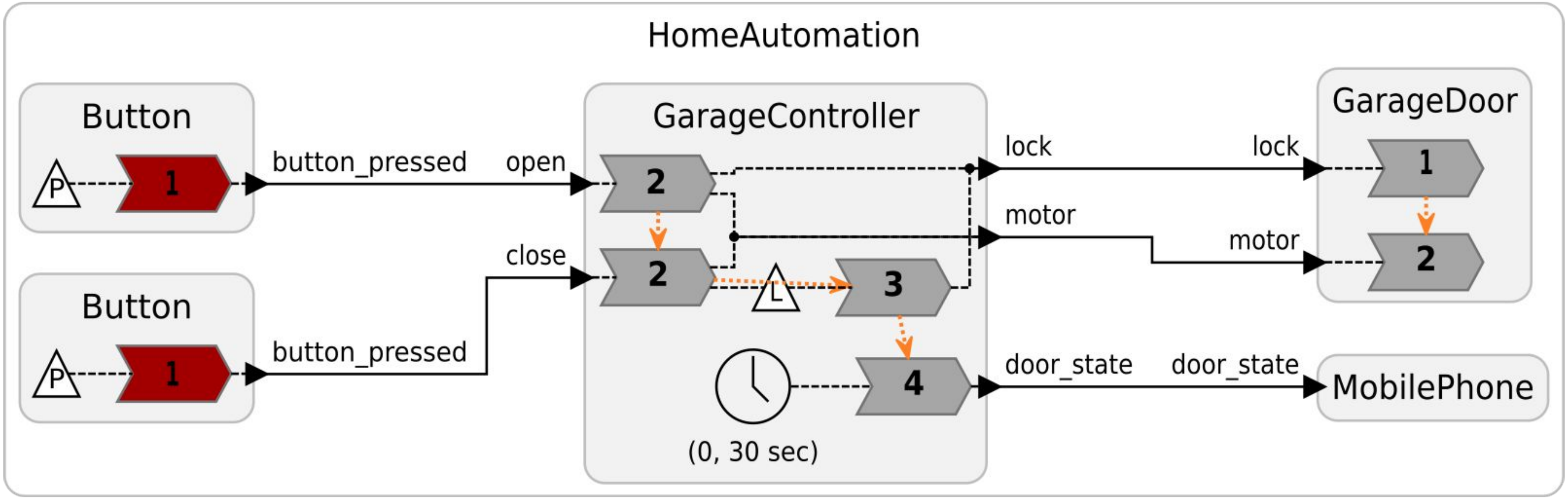


- Logical Execution Time (LET) ⇒ (tick, micro-step)
  The micro-step is required to order events happening events at the same time.
- LF only* expresses timing-semantics (Coordination).
- Reaction bodies (business logic) can be written in different target languages: C, C++, Rust, Python, TS

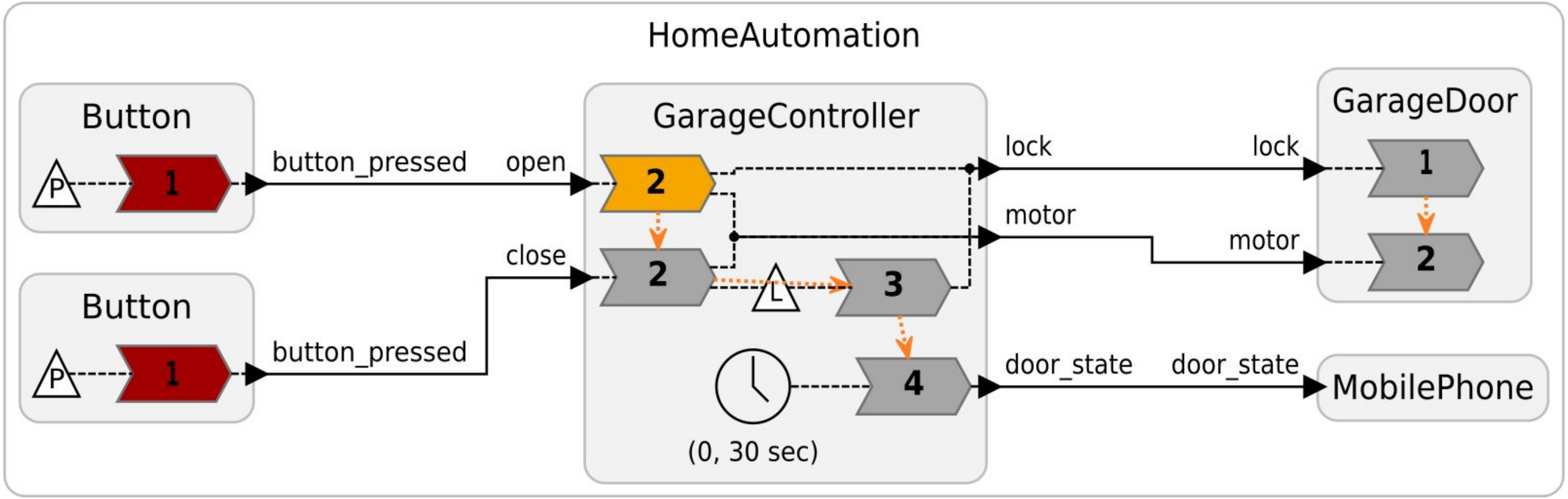
Lingua Franca



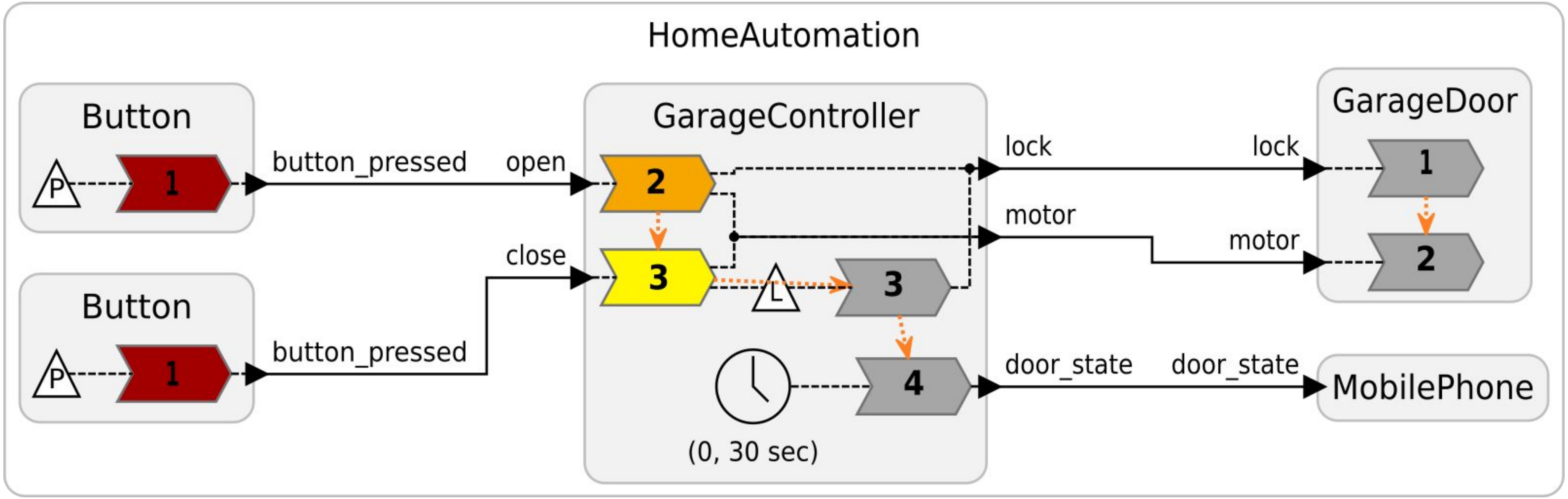
Ensuring Deterministic Execution



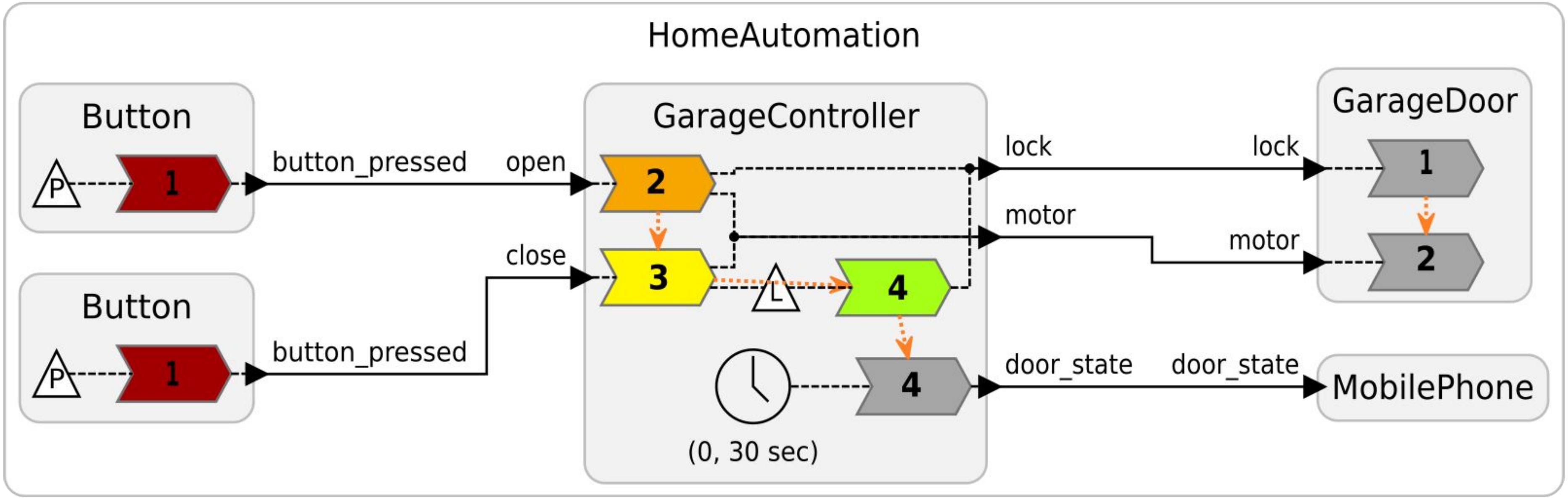
Ensuring Deterministic Execution



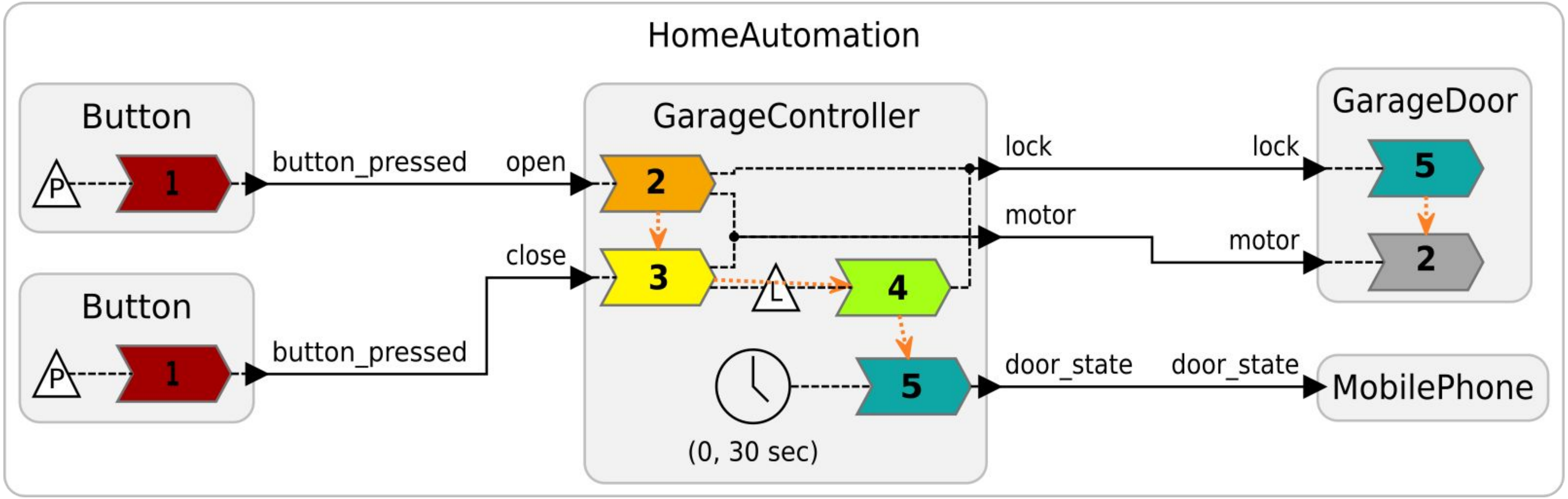
Ensuring Deterministic Execution



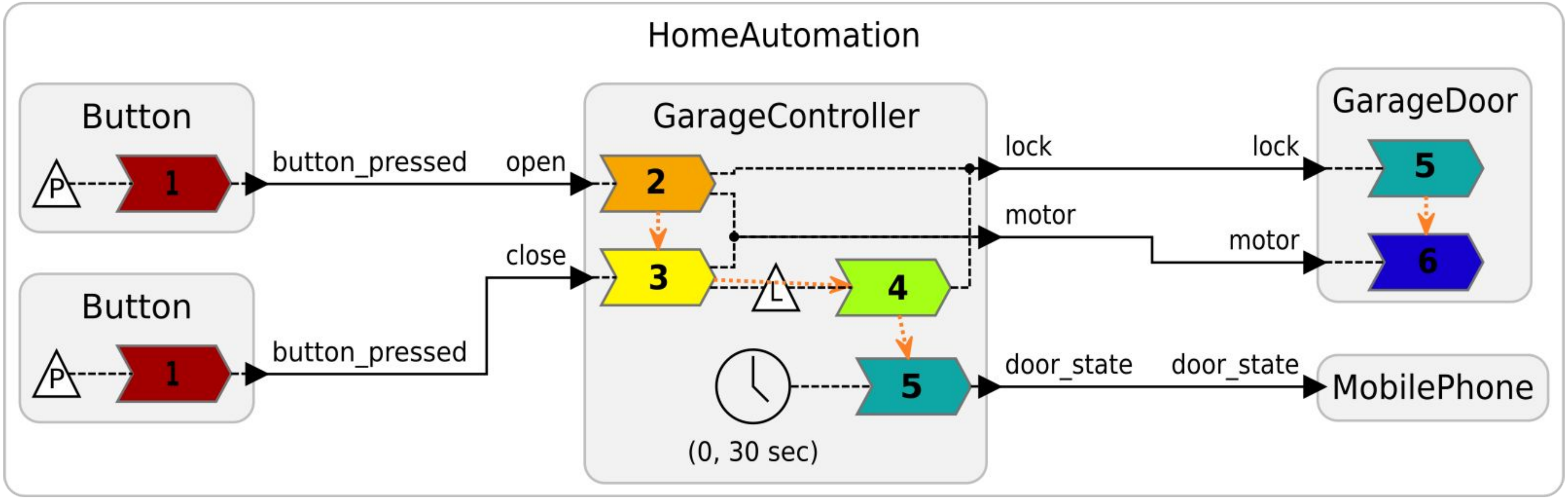
Ensuring Deterministic Execution



Ensuring Deterministic Execution



Ensuring Deterministic Execution

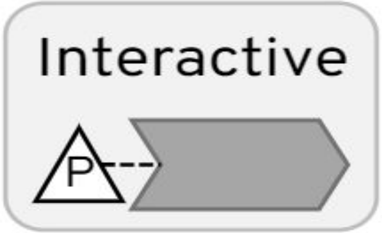


Lingua Franca is a specification.

We specify in logical time what should happen, and the compiler and runtime try to execute it to specification.

Explicit Non-Determinism

```
reactor Interactive {
    physical action a;
    reaction(a) {= =}
}
```



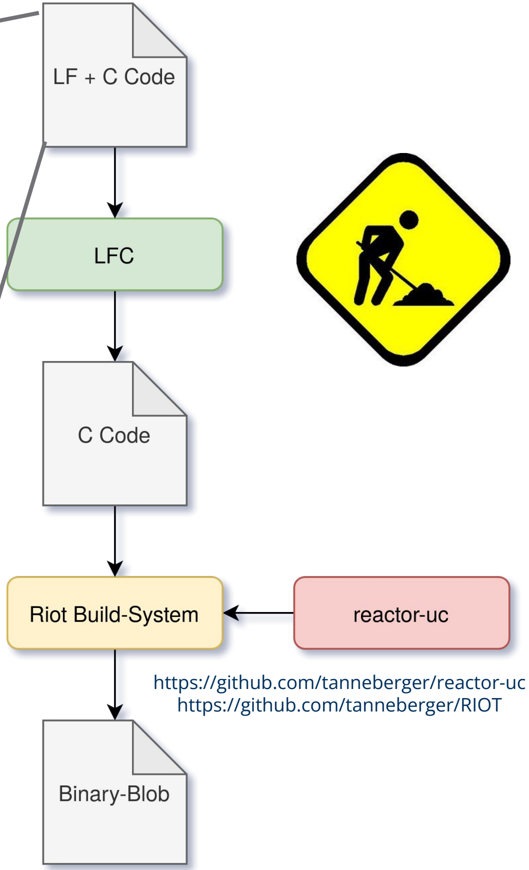
Handle Deviations

```
reactor Deadline {
    input in:int
    reaction(in) {= =} deadline (50 msec) {=
        // Deadline miss handler here
    =}
}
```

Okay, so how can I use it?

- **Shiny new runtime designed for embedded devices and use-cases.**
- **Slowly becoming feature complete.**

```
target C;

reactor GarageController {
  input open: void;
  input close: void;

  output motor: int;
  output lock: void;
  output send_door_state: int;

  logical action delay: void;
  timer t(0, 30sec);
  state door_state: int = 0;

  reaction (open) -> lock, motor {=
    lf_set(lock);
    lf_set(motor, 1);
    self->door_state = 0; // door open
  =}

  reaction (close) -> motor, delay {=
    lf_set(motor, 0);
    lf_schedule(delay, 90sec);
  =}

  reaction (delay) -> lock {=
    lf_set(lock);
    self->door_state = 1; // door closed
  =}

  reaction(t) -> send_door_state {=
    lf_set(send_door_state, self->door_state);
  =}
}
```



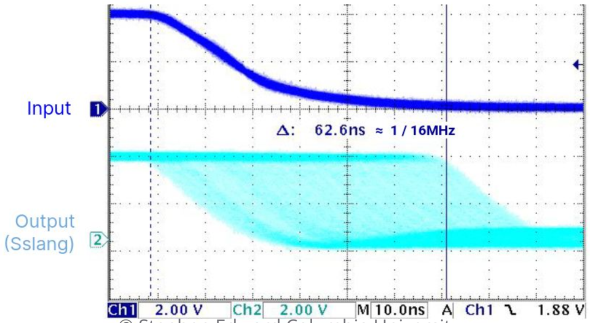
LF-RIOT Outlook

Federated execution over low-bandwidth Protocols

- Top-Down design process of IoT and Cloud-Software
- Interesting research in topics how to ensure continued execution with unstable network links.

Hardware & OS support for physical actions.

- The more accurate a time-stamp of a physical event is the more deterministic, will be the execution.
- OS Interrupts as triggers for physical actions.



Conclusions

1. Order matters!
2. Non-Determinism is very hard to debug!
3. Lingua Franca is a deterministic execution model.
4. RIOT OS support is coming very soon.

Hackathon

Join us on Saturday!

Thank you!

Backup Slides

Why Determinism

Check for updates

Determinism

EDWARD A. LEE, University of California, Berkeley

This article is about deterministic models, what they are, why they are useful, and what their limitations are. First, the article emphasizes that determinism is a property of models, not of physical systems. Whether a model is deterministic or not depends on how one defines the inputs and behavior of the model. To define behavior, one has to define an observer. The article compares and contrasts two classes of ways to define an observer, one based on the notion of "state" and another that more flexibly defines the observables. The notion of "state" is shown to be problematic and lead to nondeterminism that is avoided when the observables are defined differently. The article examines determinism in models of the physical world. In what may surprise many readers, it shows that Newtonian physics admits nondeterminism and that quantum physics may be interpreted as a deterministic model. Moreover, it shows that both relativity and quantum physics undermine the notion of "state" and therefore require more flexible ways of defining observables. Finally, the article reviews results showing that sufficiently rich sets of deterministic models are incomplete. Specifically, nondeterminism is inescapable in any system of models rich enough to encompass Newton's laws.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Computing methodologies** → **Concurrent computing methodologies**; • **Software and its engineering** → **Extra-functional properties**;

Additional Key Words and Phrases: Concurrency, determinism, distributed computing

ACM Reference format:
Edward A. Lee. 2021. Determinism. *ACM Trans. Embed. Comput. Syst.* 20, 5, Article 38 (May 2021), 34 pages.
https://doi.org/10.1145/3453652

1 INTRODUCTION

For most of my professional research career, I have sought more deterministic mechanisms for solving various engineering problems. My focus has always been on systems that combine the clean and neat world of computation with the messy and unpredictable physical world. Why the obsession with deterministic mechanisms? My wife, who is an expert in stochastic models, gives me a hard time about this obsession, observing, correctly, that deterministic models are just a special case. Why not, then, focus on the more general set of nondeterministic models?

Today, our society relies heavily on deterministic engineered systems. The balances in our bank accounts are a consequence of the inputs to the bank's computing systems. The email received is the email that was sent. The files on our computers contain the data that we put there. Our cars

Some of the work in this paper was supported by the National Science Foundation (NSF), award #CNS-1836601 (Reconciling Safety with the Internet) and the iCyPhy (Industrial Cyber-Physical Systems) research center, supported by Denso, Siemens, and Toyota.

Author's address: E. A. Lee, University of California, Berkeley, 545Q Cory Hall, Berkeley, CA, 94720-1770; email: eal@berkeley.edu.

1539-9087/2021/05-ART38
https://doi.org/10.1145/3453652

Edward A. Lee. 2021. Determinism. ACM Trans. Embed. Comput. Syst. 20, 5, Article 38 (September 2021), 34 pages. https://doi.org/10.1145/3453652

OS Synchronisation

```cpp
class SharedState {
private:
    int state = 1;
public:
    void add_one() {
        state += 1;
    }

    void multiply_two() {
        state *= 2;
    }

    auto get_state() const -> int {
        return state;
    }
};
```

OS Synchronisation

```
class SharedState {
private:
    std::mutex mutex_;
    int state = 1;
public:
    void add_one() {
        std::lock_guard<std::mutex> m(mutex_);
        state += 1;
    }

    void multiply_two() {
        std::lock_guard<std::mutex> m(mutex_);
        state *= 2;
    }

    auto get_state() -> int {
        std::lock_guard<std::mutex> m(mutex_);
        return state;
    }
};
```

OS Synchronisation

```cpp
class SharedState {
private:
    std::mutex mutex_;
    std::vector<int> events_;
    int state = 1;
public:
    void add_one() {
        std::lock_guard<std::mutex> m(mutex_);
        events_.push_back(Event::Add);
    }

    void multiply_two() {
        std::lock_guard<std::mutex> m(mutex_);
        events_.push_back(Event::Mul);
    }

    auto process() -> int {
        std::lock_guard<std::mutex> m(mutex_);

        if (std::find(events_.begin(), events_.end(), Event::Add) != std::end(events_)) {
            state += 1;
        }

        if (std::find(events_.begin(), events_.end(), Event::Mul) != std::end(events_)) {
            state *= 2;
        }

        return state;
    }
}
```