Generalizing Logical Execution Time

Edward A. Lee[0000−0002−5663−0584] and Marten Lohstroh[0000−0001−8833−4117]

UC Berkeley
Berkeley, CA, USA
eal@berkeley.edu, marten@berkeley.edu

Abstract. In the Logical Execution Time (LET) principle, concurrent software components interact deterministically, reading their inputs atomically at the start of a task and producing outputs atomically after a fixed elapsed logical time. In addition to deterministic concurrency, LET programs yield more deterministic timing when they interact with their physical environment through sensors and actuators. This paper shows through a series of examples that the LET principle can be realized flexibly and generalized using the Lingua Franca coordination language.

Keywords: Concurrent Software · Distributed Systems · Logical Execution Time.

1 Motivation

The Logical Execution Time (LET) principle was pioneered by Tom Henzinger and Christoph Kirsch (with, as always, significant contributions from others), who demonstrated its efficacy and realizability for the design of cyber-physical systems (CPSs) well before the term CPS had been coined [12]. The Giotto programming language, introduced in the very first EMSOFT conference [11] (of which Henzinger and Kirsch were founders), elegantly realized the LET principle in the form of a coordination language, where the business logic of programs was realized in a conventional language (such as C), but the modal behavior, concurrency, and timing were orchestrated by a runtime engine that closely followed the LET principle. This work inspired quite a bit follow-up work, including applications to distributed real-time automotive software [9] and automotive multicore software [3,8]. The LET principle has also been applied to programming time-predictable multicore processors [15], has been used to facilitate parallel execution of legacy software on multicore [29], and has been leveraged for schedulability analysis [14]. Whereas in Giotto execution of components is time driven, the language extensions in xGiotto support asynchronous events [10]. The Timing Definition Language (TDL) applies the LET principle in the context of Matlab/Simulink models [28].

According to Henzinger, et al., the LET principle enables "abstract, platform-independent real-time programming," and is an important step toward separating "reactivity from schedulability" [12,13]. They say,

in Principles of Systems Design - Essays Dedicated to Thomas A. Hensinzer on the Occasion of his 60th Birthday, Krishnendu Chaterjee, Laurent Doyen, Rupak Mujumdar, and Jean-Francois Raskin, Editors, LNCS 13660, Springer, 2023.

> The term reactivity expresses what we mean by control-systems aspects: the system's functionality, in particular, the control laws, and the system's timing requirements. The term schedulability expresses what we mean by platform-dependent aspects, such as platform performance, platform utilization (scheduling), and fault tolerance. Giotto decomposes the development process of embedded control software into high-level real-time programming of reactivity and low-level real-time scheduling of computation and communication. Programming in Giotto is real-time programming in terms of the requirements of control designs, i.e., their reactivity, not their schedulability. [12]

In this paper, we focus on this separation. Reactivity specifies what the designer intends to achieve, whereas schedulability specifies how an execution platform achieves that intent. We begin in Section 2 by interpreting this separation as distinct uses of models. In Section 3, we review the LET principle. In Section 4, we provide a formalism for logical and physical timelines. In Section 5, we briefly introduce the Lingua Franca coordination language, and then, in Section 6, we give a series of examples of Lingua Franca programs that flexibly apply the LET principle, allowing, for example, mitigation of the data age problem [4]. We make some concluding remarks in Section 7.

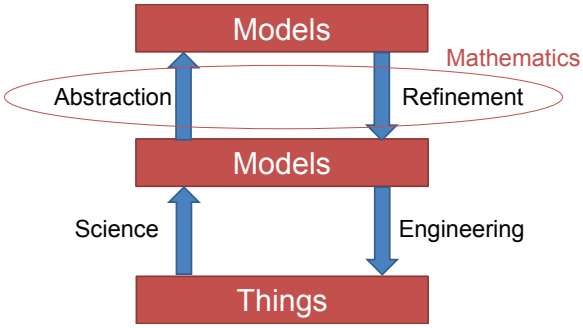

Fig. 1. Relations between models and between models and things.

2 Science, Engineering, and Mathematics

A Giotto program, and indeed any computer program that forms part of a cyber-physical system, is a **model** for the behavior of the electronic part of the system. A microcontroller, with electrons sloshing around inside, is a "thing-in-itself" (to use Kant's term), yet it is expected to behave as specified by programs it executes.

In *Plato and the Nerd*, one of us (Lee) makes a distinction between an **engineering model**, where the thing-in-itself is expected to behave like the model,

and a **scientific model**, where the model is expected to behave like the thing-in-itself [20]. These mirror-image relationships are depicted in Fig. 1. While science is concerned with establishing models that capture characteristics of physical things, the goal of engineering is to craft physical things that share properties with models. The disciplines of science and engineering are in a symbiotic relationship with one another; designs are often evaluated through scientific experimentation, and many scientific experiments are enabled by carefully engineered tools.

Models also have utility beyond their relationship to things. Mathematics treats the properties of models and relationships between models unhindered by constraints of physical realizability. In mathematics, **abstraction** is used to derive simpler models from more elaborate ones, and **refinement** is used to elaborate simpler models by adding more specificity. The operations of abstraction and refinement are key in model-based system design, an idea that has been codified in the theory of contracts [2].

When Henzinger, et al., say that Giotto specifies reactivity, what they mean is that the Giotto program serves as an engineering model (as opposed to a scientific model). It is incumbent on the compiler and the execution platform to deliver the timing and concurrency behavior that is specified by the program. Determining whether a particular execution platform can deliver the specified behavior requires a certain amount of scientific modeling, for example to find worst-case execution time (WCET) [31] bounds for tasks that are scheduled by the Giotto runtime. When Henzinger, et al., talk about schedulability, they are concerned with scientific modeling, building models of the behavior of the platform, the thing-in-itself. Reactivity, therefore, is concerned with the bottom right of Fig. 1, whereas schedulability is more concerned with the bottom left.

Once we understand this separation, it becomes natural to generalize the LET principle by strengthening the separation. In this paper, we show how to do that.

3 LET and Giotto

Software for cyber-physical systems is inevitably concurrent and timing sensitive. The time of its interactions with its physical environment is important for determining system behavior, and, unlike many information processing tasks, the goal is not simply to finish as quickly as possible. Concurrency inevitably arises because of the need to react to a multiplicity of sensors and to drive a multiplicity of actuators, and the timing of the stimulus from the sensors and the actuations cannot be arbitrary.

Traditional methods for handling concurrency and timing in software are difficult to make deterministic [17]. Consider Toyota's unintended acceleration case, where in the early 2000s, there were a number of car accidents involving Toyota vehicles that appeared to suffer from unintended acceleration. The US Department of Transportation contracted NASA to study Toyota software to determine whether software was capable of causing unintended acceleration.

The NASA study [27] was unable to find a definitive cause, but they indicted the software architecture [16]. The software used a style of design that tolerates a seemingly innocuous form of nondeterminism. Specifically, many state variables, representing for example the most recent readings from a sensor, were accessed unguarded by a variety of threads. This style of design appeals to control-system engineers because it always uses the most recent value of a sensor or shared variable, minimizing "data age" [4]. But the result is a very large number of possible behaviors that manifest nondeterministically in the field.

Logical execution time (**LET**) is a principle that delivers both deterministic concurrency and more controllable timing of the interactions with the physical components of the system. However, these two properties, deterministic concurrency and controllable timing, tend to be lumped together. We contend here that interactions between a software component and another software component should be distinguished from interactions between a software component and a physical component. Separating these two types of interaction helps to generalize the LET principle and expand its applicability.

In a LET design, the interaction between software components is defined by a logical timing model, where each task behaves as if it reads its inputs instantaneously at the start of its execution period and writes its output instantaneously after a prespecified amount of logical time has elapsed. If the inputs to the task are coming from a physical component, then the logical time of the start of execution should align reasonably precisely with some local measure of physical time. Similarly, if the outputs from the task are driving actuators, then aligning the logical time of the task completion with the physical time of actuation results in much more precisely controlled timing than we would get if we simply drive the actuator whenever the task completes.

However, if the inputs to a task are coming from another *software* component, or the outputs are going to another software component, there is no need to align these logical times with physical time as long all interactions occur in the *order* specified by their logical timing. For such interactions, for example, a logical execution time of *zero* becomes reasonable and realizable. Synchronous-reactive languages [1] are based on a hypothesis of zero execution time.

In this paper, we observe that the physical timing of interactions between software components is not an important feature of their interaction. Timing of software only matters when interaction is with the physical environment through sensors and actuators. For the interaction between software components, what really matters is determinism, not timing. Controlling their timing is one way to achieve determinism, but it is not the only way. We will give a generalization to LET that preserves determinism but reduces the use of physical timing for governing interactions between software components. Physical timing comes into play only when interacting with the physical world through timers, sensors, and actuators.

4 Logical and Physical Time

In their papers, Henzinger, et al., do not go as far as they could in separating logical and physical time. There are good reasons for this. First, every embedded system designer has a strong intuitive understanding of time, usually governed by the Newtonian model, where time is a continuum and "now" is a pointer into that continuum that advances smoothly and uniformly everywhere.

Here, we will take a different stance and distinguish logical time (a semantic property of programs) from physical time (a measurement by a physical clock). We will insist that the only access to physical time is through imperfect measurements realized by physical clocks. Newtonian time is not available to us. Logical time and physical time will be expected to align at well-chosen points in the execution of programs, only at those points, and only imperfectly.

We are interested in times of events and time intervals between events. A **physical time** $T \in \mathbb{T}$ is an imperfect measurement of time taken from some clock somewhere in the system. The set $\mathbb{T}$ contains all the possible times that a physical clock can report. We assume that $\mathbb{T}$ is totally ordered and includes two special members: $\infty \in \mathbb{T}$ is larger than any time any clock can report, and $-\infty \in \mathbb{T}$ is smaller than any time any clock can report. For example, $\mathbb{T}$ could be the set of integers $\mathbb{Z}$ augmented with the two infinite members.

Given any $T_1, T_2 \in \mathbb{T}$, the **physical time interval** (or just **time interval** if there is no ambiguity) between the two times is written $i = T_1 - T_2$. Time intervals are assumed to be members of a group $\mathbb{I}$ with a largest member ∞ and smallest member $-\infty$ and a commutative and associative addition operation. For example, $\mathbb{I}$ could be the set of integers $\mathbb{Z}$ augmented with the two infinite members. Addition involving the infinite members behaves in the expected way in that for any $i \in \mathbb{I} \setminus \{\infty, -\infty\}$,

$$\begin{aligned} i + \infty &= \infty \\ i + (-\infty) &= -\infty. \end{aligned}$$

We also assume that addition of infinite intervals saturates, as in

$$\begin{aligned} \infty + \infty &= \infty \\ (-\infty) + (-\infty) &= -\infty \\ \infty + (-\infty) &\quad \text{is undefined.} \end{aligned}$$

Note that we use the same symbols ∞ and $-\infty$ for the special members of both the set of physical times $\mathbb{T}$ and the set of intervals $\mathbb{I}$. We hope this will not create confusion.

Intervals can be added to a physical time value, and we assume that this addition is associative. I.e., for any $T \in \mathbb{T}$ and any $i_1, i_2 \in \mathbb{I}$,

$$T + (i_1 + i_2) = (T + i_1) + i_2 \in \mathbb{T}. \tag{1}$$

Addition of infinite intervals to a time value saturates in a manner similar to addition of infinite intervals.

These idealized requirements for physical times and time intervals can be efficiently approximated in practical implementations. First, it is convenient to have the set $\mathbb{T}$ represent a common definition of physical time, such as Coordinated Universal Time (UTC) because, otherwise, comparisons between times will not correlate with physical reality. In the Lingua Franca language that we use in Section 6, $\mathbb{T}$ and $\mathbb{I}$ are both the set of 64-bit integers. A $T \in \mathbb{T}$ is a POSIX-compliant representation of time, where T represents the number of nanoseconds that have elapsed since midnight, January 1, 1970, Greenwich mean time. In the Lingua Franca realization, the largest and smallest 64-bit integers represent ∞ and $-\infty$, respectively, and addition and subtraction respect the above saturation requirements. Note, however, the set of 64-bit integers is not the same as the set $\mathbb{Z}$ because it is finite. As a consequence, addition can overflow. In Lingua Franca, such overflow saturates at ∞ or $-\infty$, and as a consequence, addition is no longer associative. For example, $T + (i_1 + i_2)$ may not overflow while $(T + i_1) + i_2$ does overflow. As a practical matter, however, this will only become a problem with systems that are running near the year 2270. Only then will the behavior deviate from the ideal given by our theory.

For *logical* time, we use an element that we call a **tag** g of a totally-ordered set $\mathbb{G}$. Each event in a distributed system is associated with a tag $g \in \mathbb{G}$. From the perspective of any component of a distributed system, the order in which events occur is defined by the order of their tags. If two distinct events have the same tag, we say that they are **logically simultaneous**. We assume the tag set $\mathbb{G}$ has an element ∞ that is larger than any other tag and another $-\infty$ that is smaller than any other tag.

In the Lingua Franca language, $\mathbb{G} = \mathbb{T} \times \mathbb{U}$, where $\mathbb{U}$ is the set of 32-bit unsigned integers representing the microstep of a superdense time system [25,5,7]. We use the term **tag** rather than timestamp to allow for such a richer model of logical time. For the purposes of this paper, however, the microsteps will not matter, and hence you can think of a tag as a timestamp and ignore the microstep. We will consistently denote tags with a lower case $g \in \mathbb{G}$ and measurements of physical time $T \in \mathbb{T}$ with upper case.

We will need operations that combine tags and physical times. To do this, we assume a monotonically nondecreasing function $\mathcal{T}: \mathbb{G} \to \mathbb{T}$ that gives a physical time interpretation to any tag. For any tag g, we call $\mathcal{T}(g)$ its **timestamp**. In Lingua Franca, for any tag $g = (t, m) \in \mathbb{G}$, $\mathcal{T}(g) = t$. Hence, to get a timestamp from a tag, you just have to ignore the microstep.

The set $\mathbb{G}$ also includes infinite elements such that $\mathcal{T}(\infty_{\mathbb{G}}) = \infty_{\mathbb{T}}$ and $\mathcal{T}(-\infty_{\mathbb{G}}) = -\infty_{\mathbb{T}}$, where the subscripts disambiguate which infinity we are referring to.

An external input from outside the system, such as a user input or query, will be assigned a tag g such that $\mathcal{T}(g) = T$, where T is a measurement of physical time taken from the local clock where the input first enters the system. In Lingua Franca, this tag is normally given microstep 0, $g = (T, 0)$.

To simplify notation, we will assume a **physical time origin** $T = 0$ when a program begins executing, and will set the logical time initially to g_0, where $\mathcal{T}(g_0) = 0$. On POSIX-compliant platforms, this is not what Lingua Franca

```
1  target L;
2  reactor ReactorClass {
3      input name:type;
4      ...
5      output name:type;
6      ...
7      state name:type(init);
8      ...
9      ... timers, actions, if any ...
10     ...
11     reaction(trigger, ...) -> effect, ... {=
12         ... code in language L ...
13     =}
14     ... more reactions ...
15 }
16 ...
17 main reactor {
18     instance = new ReactorClass();
19     ...
20     instance.name -> instance.name;
21     ...
22 }
```

Fig. 2. Structure of a Lingua Franca program for target language *L*.

does. Instead, physical time is the Unix epoch time, the number of nanoseconds that have elapsed since January 1, 1970. Those numbers, however, are difficult to read, so we will give all times relative to the start of program execution.

5 Introduction to Lingua Franca

Lingua Franca (or LF, for short)[1] is a coordination language developed jointly at UC Berkeley, TU Dresden, UT Dallas, and Kiel University [24]. Applications are defined as concurrent compositions of components called **reactors** [22,21]. Fig. 2 outlines the structure of a Lingua Franca program. One or more **reactor classes** are defined with **input ports** (line 3), **output ports** (line 5), **state variables** (line 7), and timers and actions. We will elaborate on actions later. Inputs are handled by **reactions**, as shown on line 11. Reactions declare their **triggers**, as on line 11, which can be input ports, timers, or actions. If a reaction lists an output port among its **effects**, then it can produce tagged output messages via that output port. The routing of messages is specified by **connections**, as shown on line 20. The syntax and semantics will become clearer as we develop our specific examples.

6 LET and More in Lingua Franca

In this section, we show through a series of examples how Lingua Franca can realize concurrent programs under the LET principle, but is also more flexible. The key to this flexibility is that Lingua Franca distinguishes logical time from

[1] https://lf-lang.org

```
reactor Sensor(p:time(10 ms)) {
    output out:int;
    timer t(0,p);
    reaction(t) -> out {=
        ... retrieve sensor data and produce it ...
    =}
}
reactor Task1 {
    input in:int;
    output out:int;
    reaction(in) -> out {=
        ... process sensor data ...
    =}
}
reactor Task2 {
    input in:int;
    output out:int;
    reaction(in) -> out {=
        ... further process sensor data ...
    =}
}
reactor Actuator {
    input in:int;
    reaction(in) {=
        ... drive actuator ...
    =}
}
main reactor(p:time(10 ms)) {
    s = new Sensor(p = p);
    t1 = new Task1();
    t2 = new Task2();
    a = new Actuator();
    s.out -> t1.in;
    t1.out -> t2.in;
    t2.out -> a.in;
}
```

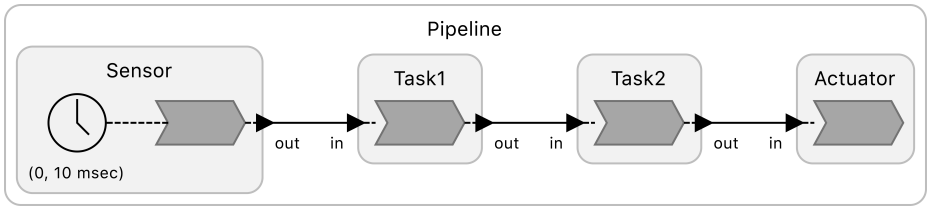

Fig. 3. Basic pipeline.

physical time and enables alignment at cyber-physical interaction points [23]. As we go through the examples, we explain in more detail how execution of Lingua Franca programs works.

6.1 Periodic Polled Control System

Consider a pipeline of tasks between a sensor and actuator shown in Fig. 3. One might find this example in the software portion of a feedback control system. The figure on the bottom is automatically generated and displayed by

the development tools.[2] The chevrons in the figure represent reactions, and their dependencies on inputs and their ability to produce outputs is shown using dashed lines.

In this example, a sensor is polled with a period given by the parameter p, which has a default value of 10 ms. By default, LINGUA FRANCA reactions are logically instantaneous, so this program is more like a synchronous-reactive program than like a LET program. The timer t produces a sequence of events with tags g_i, $i = 0, 1, 2, \cdots$, where $\mathcal{T}(g_i) = 10i$ ms. The runtime system first advances its **current tag** to g_0 and executes all reactions that are triggered at that tag with ordering constraints implied by data dependencies. In this example, it executes the Sensor reaction, and if that reaction produces an output, then it will execute the Task1 reaction. If Task1 produces an output, it will then execute Task2, and finally, if Task2 produces an output, it will execute Actuator. All of these executions will occur at tag g_0, and all will complete before the runtime advances its current tag to g_1.

Note that unlike a LET program, there is no parallelism in this program. Task2 cannot begin executing until Task1 has completed. Moreover, the logical time $\mathcal{T}(g_0)$ of the actuation is *the same* as the logical time of sensing, which would not be the case with LET. The physical time at which the actuation occurs will be determined by the execution times of the tasks, again a feature one would not find in a LET design.

```
1 main reactor(p:time(10 ms)) {
2     ...
3     t1.out -> t2.in after p;
4     t2.out -> a.in after p;
5 }
```

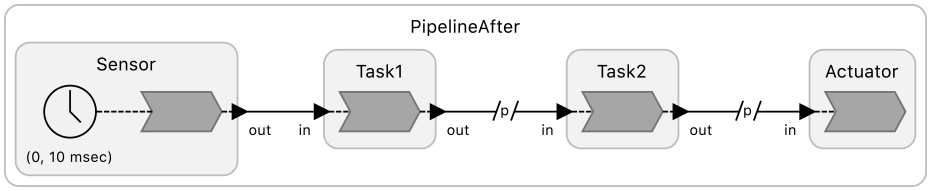

Fig. 4. Pipeline with logical delays, emulating LET.

In Fig. 4 we modify the last two lines of the program in Fig. 3, thereby converting this program to use the LET principle. The syntax "`after p`" specifies that the output produced by `t1.out` at tag g should be received by `t2.out` with tag g', where $\mathcal{T}(g') = \mathcal{T}(g) + p$. This has several consequences.

[2] The diagram synthesis feature was created by Alexander Schulz-Rosengarten of Kiel University using the graphical layout tools from the KIELER Lightweight Diagrams framework [30] (see https://rtsys.informatik.uni-kiel.de/kieler).

First, Task1 and Task2 can execute in parallel, exploiting a multicore architecture. While Task1 is handling sensor data at tag g_i, for $i \geq 1$, Task2 is processing its previous result computed with the sensor data from g_{i-1}.

Second, the latency between sensing and actuation is more constant now and less dependent on execution time. Assuming that Task1 and Task2 each are able to complete within time p, the runtime system will advance its current tag to g_i at physical time $T_i \geq \mathcal{T}(g_i)$, but T_i will be very close to $\mathcal{T}(g_i)$ because the system will have gone idle prior to that physical time. Hence, the physical latency between sensing and actuation will be close to 20 ms with the default value for the parameter $p = 10$ ms.

Compared to Fig. 3, the data delivered to the Actuator is based on older sensor input, so the designer is faced with a tradeoff between data age [4] and predictable, repeatable timing. In many safety-critical systems, repeatable timing is extremely valuable; for one, it greatly enhances the value of testing [26,18].

Assuming all reactions produce outputs, at each tag g_i for $i \geq 2$, there are three computations that can proceed in parallel. The first is to invoke the Sensor reaction followed by Task1, the second is to invoke Task2, and the third is to invoke the Actuator. If there are at least three cores, then they can all execute in parallel. If there are fewer than three cores, however, we may wish to prioritize the execution the Actuator reaction so that actuation occurs as closely as possible to 20 ms after sensing. In Lingua Franca, a simple way to do this is to assign a deadline to the reaction of the Actuator, as shown in Fig. 5. The Lingua Franca runtime uses an earliest-deadline-first (EDF) scheduling policy, and hence, the mere presence of a deadline ensures that the Actuator reaction will execute before the others.

```
...
reactor Actuator {
    input in:int;
    reaction(in) {=
        ... drive actuator ...
    =} deadline (1 ms) {=
        ... handle a deadline miss ...
    =}
}
main reactor(p:time(10 ms)) {
    ...
    t1.out -> t2.in after p;
    t2.out -> a.in after p;
}
```

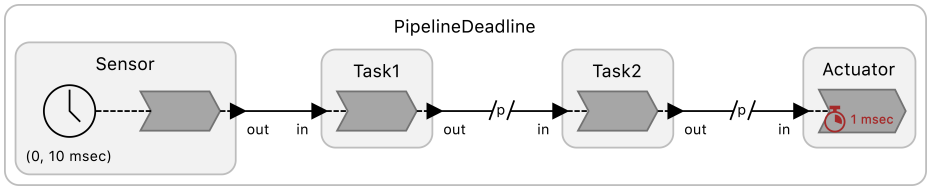

Fig. 5. Pipeline with deadline.

In addition, the deadline construct provides a **fault handling** mechanism. Line 6 in Fig. 5 specifies a deadline $d = 1$ ms. The meaning of this specification is that if the physical time T at which the runtime system invokes the reaction to an input with tag g is larger than $\mathcal{T}(g)$ by more than d, i.e. $T > \mathcal{T}(g) + d$, then a **deadline miss** has occurred, and the runtime system will invoke the code on line 7 rather than the code on line 5.

There is a subtle difference between these Lingua Franca pipelines and the LET principle as realized in Giotto. The reactions in Lingua Franca are still logically instantaneous even if their outputs are subjected to a logical delay using the `after` keyword. In Lingua Franca, an input or output port is modeled as a function $P\colon \mathbb{G} \to V \cup \{\epsilon\}$, where V is a set of **values** (a data type) and ϵ represents **absent**, the absence of a value. Because P is a function, at each tag g, a port cannot have more than one value. Since reactions are logically instantaneous, therefore, input values do not change during their execution, exactly as in LET. But any output values that are produced during that execution have the same tag as the input that triggered them. This is why downstream reactions have to be executed after completion of upstream reactions if the connection has no logical delay, like the connection between Sensor and Task1. Only then is the input to the downstream reaction known.

6.2 Federated Execution

When executing on a single machine, the current Lingua Franca runtime system completes execution of all reactions at any given tag g before advancing its **current tag** to some $g' > g$. In effect, this imposes a **barrier synchronization** between threads that might be executing in parallel on multiple cores (see Section 6.3 for our proposed extension that relaxes this barrier synchronization). The "after" clauses in Fig. 4 enable parallel execution by making available multiple input events to distinct reactors at the same tag.

Another way to achieve parallel execution in Lingua Franca is to remove the barrier synchronization and allow reactors to maintain separate and distinct current tags. This requires that messages between reactors be queued and explicitly tagged so that each reactor can process events in tag order. This can be accomplished by declaring the top-level reactor to be a **federated reactor**, as shown in Fig. 6.[3] When the top-level reactor is declared to be federated, the Lingua Franca code generator produces a separate program for each reactor instantiated within that top level. To get the same degree of parallelism as in Fig. 4, Fig. 6 creates two intermediate reactors called "Bundle1" and "Bundle2." The code generator will produce two programs, one for each Bundle. Each of these programs maintains its own current tag, and, as a consequence, Bundle2 can be processing an earlier tag while Bundle1 is processing a later one. A third program, a **runtime infrastructure** (RTI), coordinates startup and shutdown and possibly mediates communication and regulates advancement of the current tag.

[3] Federated execution of Lingua Franca was largely created by Soroush Bateni.

```
...
reactor Bundle1 {
    output out:int;
    s = new Sensor(p = p);
    t1 = new Task1();
    s.out -> t1.in;
    t1.out -> out;
}
reactor Bundle2 {
    input in:int;
    t2 = new Task2();
    a = new Actuator();
    in -> t2.in;
    t2.out -> a.in;
}
federated reactor(p:time(10 ms)) {
    b1 = new Bundle1();
    b2 = new Bundle2();
    b1.out -> b2.in;
}
```

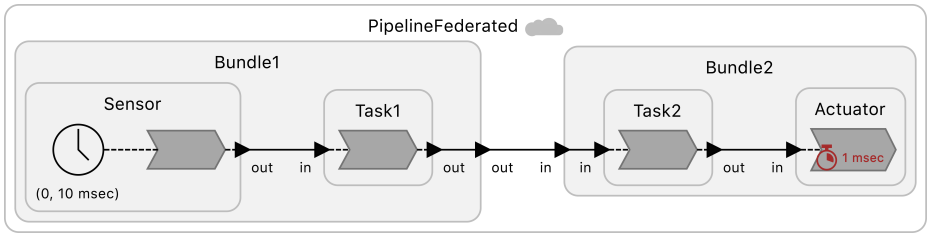

Fig. 6. Federated pipeline.

Lingua Franca provides two distinct mechanisms for coordinating the execution of federated reactors [19]. The default mechanism is **centralized**, where each federate (each Bundle) consults with the RTI before advancing its current tag. This mechanism makes use of knowledge of the interconnection topology. In this example, there are no cycles in the communication pattern, and, consequently, Bundle1 can advance its current tag with no constraints. It has no inputs that might later see messages with earlier tags. Bundle2, however, cannot advance to tag g until it has been assured by the RTI that it has seen all inputs with tags less than g.

In **decentralized** coordination, federates rely on clock synchronization and assumed bounds on communication latencies and advance their current tag to g when their physical clock has advanced sufficiently that, given these assumptions, they have seen all inputs with tags less than g. This principle has been previously used in PTIDES [32] and Google Spanner [6]. For details, see Lee, et al. [19].

In Fig. 6, there are no "after" logical delays, so the timing of actuation relative to sensing will depend on execution times of the task and communication latency between bundles. The latency is bounded by a specified deadline. A single "after" delay on the connection between Task2 and Actuator would be sufficient to get the effect of a logical execution time. But now, this LET represents

```
...
main reactor(p:time(10 ms)) {
    ...
    timer t(0, 1 ms);
    reaction(t) {=
        // ... handle quick reaction ...
    =} deadline(500 usec) {=
        // ... handle deadline violation ...
    =}
}
```

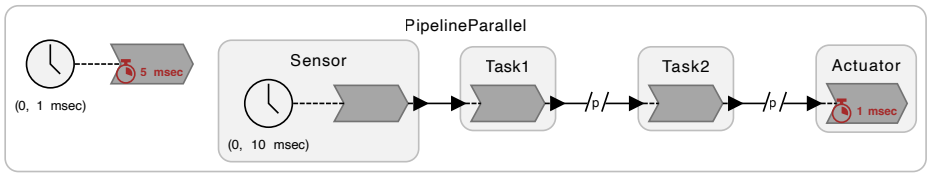

Fig. 7. Variant of Fig. 5 with an additional periodic task.

the logical execution time of the entire pipeline from Sensor to Actuator. We have effectively decoupled the timing of interactions between the cyber and the physical parts from the timing (and parallelism) of the interactions between the cyber parts.

6.3 Relaxing the Barrier Synchronization

Consider the variant of Fig. 5 shown in Fig. 7. The only change is the addition of one more reaction that reacts to a timer with a period of one millisecond. When executed on a single machine with one or more cores, the current implementation of the Lingua Franca runtime, with its barrier synchronization, has a major difficulty with this program. In this section, we show how to use the LET principle to eliminate this difficulty.

First, we explain the difficulty with the current runtime system. The reaction shown at the left in the figure is triggered at intervals of one ms and has a deadline of 500 microseconds. The reactions in the pipeline are triggered at intervals of 10 ms, and actuation has a deadline of one ms. At the one-out-of-ten reactions where these periodic events align, tags with 0, 10, 20, ... ms, the unnamed reaction at the left will be given priority over any other reaction because of its tighter deadline. So far so good. But what if the reactions in Task1 or Task2 take more than 1.5 ms to execute? What happens at tags with 1, 11, 21, ... ms? With the barrier synchronization, the execution of the Task1 and Task2 reactions will prevent the advancement of time to 1, 11, 21, ... and will thereby cause a deadline violation in the invocation of the unnamed reaction at those tags!

We could use federated execution to eliminate this problem, but, in practice, a new difficulty will arise because thread priorities will now have to be coordinated across *processes*, not just threads within a process. Moreover, thread scheduling introduces additional inefficiencies because of the need to send data

across processes and the added overhead of coordinating the advancement of tags across processes. It would be better to solve this problem within a single multithreaded process.

We can use the LET principle to eliminate this problem within a single process. In particular, the reactions in Task1 and Task2 have the property that their effects (the outputs they produce) are all delayed by $p = 10$ ms. As a consequence, these two reactions can be temporarily withdrawn from the barrier synchronization to rejoin it only when the tag is to advance to the next 10 ms period. Hence, the tag can be advanced to 1, 11, 21, ... even though these reactions at tags with 0, 10, 20, ... have not yet completed. In a multithreaded execution, even if there is only a single core, the Task1 and Task2 reactions can proceed concurrently and can be preempted by a thread that is to execute the unnamed reaction, thereby avoiding the deadline violation. All that is required is that the underlying thread scheduler respect priorities, and that priorities be assigned to worker threads according to the deadlines of the reactions assigned to them.

The general principle is simple. For any reaction that produces outputs, it can be treated as a LET task with the LET equal to the minimum **after** delay of all of its outputs. The specific treatment is that if the LET is greater than zero, then the worker thread executing the reaction need not participate in the barrier synchronization until the time comes to advance the tag to t + LET. That worker thread can continue executing at logical time t while the rest of the program advances its logical time.

With this enhancement, we claim that Lingua Franca will be capable of everything a classical LET system can do. But it can also do more, a property that becomes obvious when we consider less regular, aperiodic executions, as we do next.

6.4 Event Triggered Execution

In the examples given so far, the sensor input is periodic, polled using a timer. A more interesting scenario arises when inputs from the physical world are events with uncontrolled timing, for example arising through an interrupt request. In Lingua Franca, such an external event is realized with a **physical action**, shown in Fig. 8, which is depicted in the diagram as a triangle with a "P". Line 4 defines the physical action and line 5 defines a reaction that reacts to the physical action. This reactor will also need some additional code (not shown) to interface to some physical device and call a built-in `lf_schedule()` function to schedule the physical action when an external event occurs. This could be done, for example, in a callback function or an interrupt service routine.

When an external event triggers a call to `lf_schedule()`, the Lingua Franca runtime system consults the local physical clock, reading from it a time T, and creates an event with tag g such that $\mathcal{T}(g) = T$. The reaction on line 5, therefore, will be invoked at tag g, and the timestamp of the tag will represent the physical time of the external event as measured by a local clock.

Notice that, now, using the "after" logical delays of Fig. 4 will *not* yield parallel execution without the enhancement described in the previous section,

```
...
reactor Event {
    output out:int;
    physical action a:int;
    reaction(a) -> out {=
        ... retrieve sensor data and produce it ...
    =}
}
main reactor(p:time(10 ms)) {
    s = new Event();
    t1 = new Task1();
    t2 = new Task2();
    a = new Actuator();
    s.out -> t1.in;
    t1.out -> t2.in;
    t2.out -> a.in;
}
```

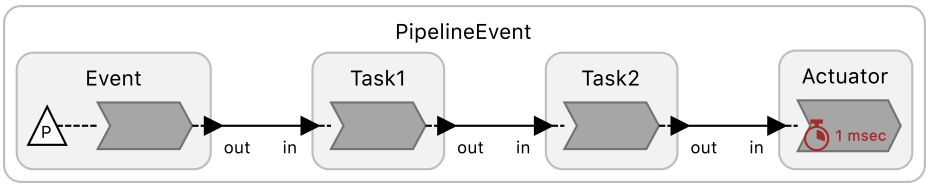

Fig. 8. Event-triggered pipeline.

6.3. This is because when a new event with tag g' arrives, it is unlikely to have timestamp $\mathcal{T}(g') = \mathcal{T}(g) + a$, where g is the tag of the previous event and a is the "after" delay. Only if that coincidence occurs can the pipelined reactions execute in parallel (in the absence of our enhancement). Without the enhancement, to get parallel execution, we will have to use a federated program like that of Fig. 6. In this case, parallel execution *is* possible because Task2 may be still processing the previous event when a new event arrives.

This situation, however, reveals a subtlety about the enhancement of Section 6.3. Suppose Fig. 8 has **after** delays of 10 ms, like Fig. 4. Suppose that Task1's reaction has withdrawn from the barrier synchronization at tag 10 ms, say, and a new event occurs at tag 12 ms. In this case, it will not do for the reaction to remain withdrawn until 20 ms because this could result in Task1's reaction being invoked at tag 12 while it is still processing the event at tag 10 in some other thread. In Lingua Franca, any two reactions belonging to the same reactor must have mutually exclusive invocations because they share state and there is no assurance that users have written reentrant code. Hence, our enhancement must be careful advancing tags while there are executing reactions that have withdrawn from the barrier synchronization. In particular, if any event at a tag t is about to trigger a reaction belonging to a reactor that has an executing reaction at tag $t' < t$, then that triggered reaction must be blocked until the executing reaction at t' completes. By blocking that reaction, we will also block any reactions that depend on it, thereby preserving determinism.

This raises an interesting and subtle semantic property of Lingua Franca that is not shared with any LET system. Specifically, logically, a reaction *always*

executes in zero time in that its local state gets updated without logical time advancing during that updating. Hence, the term "logical execution time" is no longer quite adequate, even though it can match the same concurrency properties of a classical LET system. In a classical LET system, the *local state* takes time to update, not just the externally visible effects, and hence, a classical LET task cannot be interrupted with a new execution during its logical execution time. In Lingua Franca, it *can* be interrupted without undermining its determinism, which is a strict generalization over LET.

Such interruptions, however, may not be a good idea in practical applications. An unconstrained physical action like that of Fig. 8 runs a risk of overwhelming the software system and disrupting timing. If `lf_schedule()` is called while an earlier event is still being processed, the new event will simply be queued to be handled when prior tags have been fully processed. This could result in an unbounded buildup of queued events, for example if the physical action is triggered by a network input and the system is under a denial-of-service attack.

Fortunately, Lingua Franca provides mechanisms to prevent such eventualities. First, a physical action can have a **minimum spacing** parameter, a minimum logical time interval between tags assigned to events. When the environment tries to violate this constraint by issuing requests too quickly, the programmer can specify one of three policies: drop, replace, or defer. The drop policy simply ignores the event. The replace policy replaces any previously unhandled event, or if the event has already been handled, defers. The defer policy assigns a tag g to the event with timestamp $\mathcal{T}(g)$ that is larger than the previous event by the specified minimum spacing.

While the minimum spacing parameter ensures that tags are sufficiently spaced, it does not, by itself, ensure that the scheduler will prioritize execution of the reaction in the Actuator reactor. We can again specify a **deadline** associated with that reaction, thereby ensuring that the Actuator reaction will execute first, resulting in greater precision in Sensor-to-Actuator latency.

We can further combine minimum spacing, deadlines, and "after" delays to maximize parallelism and timing precision under overload conditions, when the physical action repeatedly triggers with the minimum spacing. The resulting program is shown in Fig. 9. Line 4 declares the physical action to have minimum delay of 0 and a minimum spacing of 10 ms (the minimum delay argument is not relevant to our discussion here). Under burst conditions, this program will experience input events every 10 ms, and after the first two such events, at each 10 ms boundary, the Actuator reaction will have top priority. If two cores are available, then one will execute Actuator followed by Task2 while the other executes Sensor followed by Task1. The latency from Sensor to Actuator will be close to 20 ms, thereby realizing the goals of LET, but now in an event-triggered system rather than a periodic one.

```
1   ...
2   reactor Event {
3       output out:int;
4       physical action a(0, 10 ms);
5       reaction(a) -> out {=
6           ... retrieve sensor data and produce it ...
7       =}
8   }
9   main reactor(p:time(10 ms)) {
10      s = new Event();
11      t1 = new Task1();
12      t2 = new Task2();
13      a = new Actuator();
14      s.out -> t1.in;
15      t1.out -> t2.in after 10 ms;
16      t2.out -> a.in after 10 ms;
17  }
```

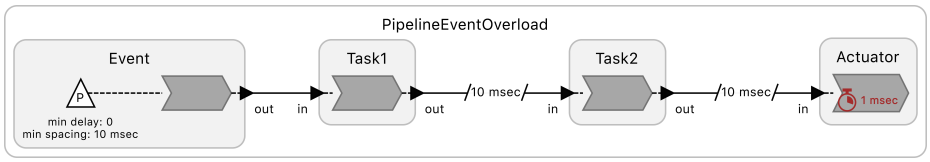

Fig. 9. Event-triggered pipeline optimized for overload conditions.

6.5 Merging Events with Periodic Tasks

A common scenario in cyber-physical systems is that asynchronous events are mixed with periodic actions. For example, a feedback control system may operate with a regular sample rate, but sporadic events may result in changes in the control laws. Lingua Franca provides mechanisms for again achieving regular, tightly controlled timing.

Fig. 10 shows an example where a **logical action** (depicted as a triangle with an "L") is used to precisely align the asynchronous events of a physical action with the periodic events of a pipeline. Here, the source code listing gives the details of the modified Event reactor as realized in the C target. This reactor accepts asynchronous events via its physical action, but then delays production of an output until the next logical time that will align with the timer driving the Sensor reactor. Specifically, the reaction defined on line 10 calculates the time interval to the next multiple of 10 ms and schedules a logical action b to trigger at that next multiple of 10 ms. Line 14 calculates a waiting time that is assured to be between 1 ns and 10 ms, where the latter value is chosen in the (unlikely) event that the physical action triggers at exactly the time of one of the timer triggers. That call to `lf_schedule()` will result in an invocation of the reaction defined on line 6 that will be precisely aligned with the next periodic sensor data, such that the reaction in Task2 will see two simultaneous inputs. The reaction in Task2 checks for the presence of an event on in1. This reaction is guaranteed to be invoked every 10 ms by this program, regardless of the timing of asynchronous inputs, thereby yielding highly deterministic timing. This design relies on the associativity of addition of time intervals.

```
target C;
reactor Event {
    output out:int;
    physical action a(0, 10 ms):int;
    logical action b:int;
    reaction(b) -> out {=
        // Produce as output previously received event.
        SET(out, b->value);
    =}
    reaction(a) -> b {=
        // Get the time assigned to the physical action.
        instant_t current_time = get_elapsed_logical_time();
        // Calculate the time to the next multiple of 10 ms.
        interval_t wait = MSEC(10) - current_time % MSEC(10);
        // Schedule a logical action to trigger an output.
        schedule_int(b, wait, a->value);
    =}
}
reactor Task2 {
    input in1:int;
    input in2:int;
    output out:int;
    reaction(in1, in2) -> out {=
        if (in1->is_present) {
            // ... react to asynchronous event ...
        } else if (in2->is_present) {
            // ... react to periodic event ...
        }
    =}
}
...
```

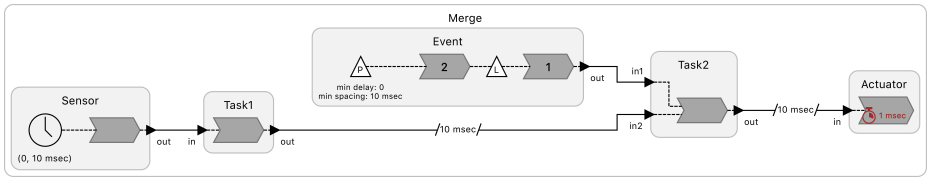

Fig. 10. Merging of asynchronous events with periodic ones.

6.6 Shared State

In Giotto, there is an assumption that tasks do not interact except through their input and output ports. In Lingua Franca, in contrast, reactions within the same reactor can share state variables. Fig. 11 shows a variant of Fig. 10 that takes advantage of this feature to realize a common pattern, where an asynchronous event changes the control law used to process periodic events.

The new version of Task2 now has two distinct reactions, one of which reacts to the asynchronous event by changing the control law, and the other of which reacts to the periodic inputs to apply the control law. On line 6, a state variable named "control_law" is defined. In Lingua Franca semantics, if the two input ports have simultaneous input events, then the first reaction executes to completion before the second reaction executes, so access to the state variable is mutually exclusive and deterministically ordered. No such ordering is en-

forced between reactions across different reactors, enabling parallel execution of logically simultaneously triggered reactions that do not share state.

Notice in Fig. 11 that we no longer need the logical action of Fig. 10. The effect of the new control law is guaranteed to align with the 10 ms timing of the periodic events.

```
...
reactor Task2 {
    input in1:int;
    input in2:int;
    output out:int;
    state control_law:int;
    reaction(in1) {=
        // Change control law.
        self->control_law = in1->value;
    =}
    reaction(in2) -> out {=
        // ... process sensor data using control_law state variable ...
    =}
}
reactor Event {
    output out:int;
    physical action a(0, 10 ms):int;
    reaction(a) -> out {=
        SET(out, a->value);
    =}
}
...
```

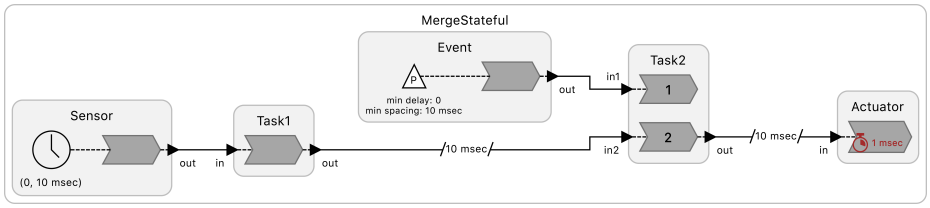

Fig. 11. Reactor with a state variable.

7 Conclusions

The LET principle accomplishes two distinct things. First it specifies the timing of the interaction between a software component and its environment, which consists of other software components and physical sensors and actuators. Second, perhaps even more importantly, it provides a deterministic concurrency model. That is, the interaction between software components does not depend on their execution time (as long as WCET bounds are respected).

We have shown that these two principles can be better separated. In Lingua Franca, the deterministic concurrency model is provided by the use of tags that realize a logical timeline. This logical timeline is decoupled from physical time

except at points where the program interacts with its physical environment by explicitly invoking timers, physical actions, and deadlines.

The result generalizes the LET principle, enabling combinations of logical execution time with the zero execution time semantics of synchronous languages while preserving the ability to precisely control the timing of interactions with the physical environment. Lingua Franca also does not restrict the use of LET to periodic systems.

Acknowledgments

The authors would like to acknowledge and thank the following people for their contributions to the design and implementation of Lingua Franca: Soroush Bateni, Peter Donovan, Clément Fournier, Hokeun Kim, Shaokai Lin, Christian Menard, Alexander Schulz-Rosengarten, Matt Weber, and Steven Wong. We also thank Libero Nigro and anonymous reviewers for constructive suggestions. The work in this paper was supported in part by iCyPhy (the Industrial Cyber-Physical Systems) research center, supported by Denso, Siemens, and Toyota.

References

1. Benveniste, A., Berry, G.: The synchronous approach to reactive and real-time systems. Proceedings of the IEEE **79**(9), 1270–1282 (1991)
2. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.G.: Contracts for System Design. Research Report RR-8147, INRIA (Nov 2012), https://hal.inria.fr/hal-00757488
3. Biondi, A., Natale, M.D.: Achieving predictable multicore execution of automotive applications using the LET paradigm. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (April 2018). https://doi.org/10.1109/RTAS.2018.00032
4. Bradatsch, C., Kluge, F., Ungerer, T.: Data age diminution in the logical execution time model. In: International Conference on Architecture of Computing Systems (ARCS). vol. LNCS 9637, pp. 173–184. Springer (2016). https://doi.org/10.1007/978-3-319-30695-7_13
5. Cataldo, A., Lee, E.A., Liu, X., Matsikoudis, E., Zheng, H.: A constructive fixed-point theorem and the feedback semantics of timed systems. In: Workshop on Discrete Event Systems (WODES) (2006)
6. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. In: OSDI (2012). https://doi.org/10.1145/2491245
7. Cremona, F., Lohstroh, M., Broman, D., Lee, E.A., Masin, M., Tripakis, S.: Hybrid co-simulation: it's about time. Software and Systems Modeling **18**, 1655–1679 (November 2017). https://doi.org/10.1007/s10270-017-0633-6

8. Ernst, R., Kuntz, S., Quinton, S., Simons, M.: The logical execution time paradigm: New perspectives for multicore systems (dagstuhl seminar 18092). Dagstuhl Reports **8**, 122–149 (2018). https://doi.org/10.4230/DagRep.8.2.122, https://hal.inria.fr/hal-01956964
9. Gemlau, K.B., Köhler, L., Ernst, R., Quinton, S.: System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software. ACM Transactions on Cyber-Physical Systems **5**(2), 1–27 (January 2021). https://doi.org/10.1145/3381847
10. Ghosal, A., Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.: Event-driven programming with logical execution times. In: Seventh International Workshop on Hybrid Systems: Computation and Control (HSCC). vol. LNCS 2993, pp. 357–371. Springer-Verlag (2004)
11. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. In: EMSOFT 2001. vol. LNCS 2211, pp. 166–184. Springer-Verlag (2001)
12. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. Proceedings of IEEE **91**(1), 84–99 (January 2003). https://doi.org/10.1109/JPROC.2002.805825
13. Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A., Pree, W.: From control models to real-time code using Giotto. IEEE Control Systems Magazine **23**(1), 50–64 (February 2003). https://doi.org/10.1109/MCS.2003.1172829
14. Hladik, P.E.: A brute-force schedulability analysis for formal model under logical execution time assumption. In: ACM Symposium on Applied Computing (SAC). pp. 609–615 (2018). https://doi.org/10.1145/3167132.3167199
15. Kluge, F., Schoeberl, M., Ungerer, T.: Support for the logical execution time model on a time-predictable multicore processor. ACM SIGBED Review **13**(4), 61–66 (September 2016). https://doi.org/10.1145/3015037.3015047
16. Koopman, P.: A case study of Toyota unintended acceleration and software safety (2014), http://betterembsw.blogspot.com/2014/09/a-case-study-of-toyota-unintended.html
17. Lee, E.A.: The problem with threads. Computer **39**(5), 33–42 (2006)
18. Lee, E.A.: Determinism. ACM Transactions on Embedded Computing Systems (TECS) **20**(5), 1–34 (July 2021). https://doi.org/10.1145/3453652
19. Lee, E.A., Bateni, S., Lin, S., Lohstroh, M., Menard, C.: Quantifying and generalizing the CAP theorem. arXiv:2109.07771 [cs.DC] (September 16 2021), https://arxiv.org/abs/2109.07771
20. Lee, E.A.: Plato and the Nerd — The Creative Partnership of Humans and Technology. MIT Press (2017)
21. Lohstroh, M.: Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems. Ph.D. thesis, EECS Department, University of California, Berkeley (Dec 2020), http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html
22. Lohstroh, M., Íncer Romeo, Í., Goens, A., Derler, P., Castrillon, J., Lee, E.A., Sangiovanni-Vincentelli, A.: Reactors: A deterministic model for composable reactive systems. In: 8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy'19). vol. LNCS 11971. Springer-Verlag (2019)
23. Lohstroh, M., Lee, E.A.: A language for deterministic coordination across multiple timelines. In: 2020 Forum for Specification and Design Languages, FDL 2020, Kiel, Germany, September 15-17, 2020. pp. 1–8. IEEE (2020)
24. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a lingua franca for deterministic concurrent systems. ACM Transactions on Embedded Computing Systems (TECS) **20**(4), Article 36 (May 2021). https://doi.org/10.1145/3448128

25. Maler, O., Manna, Z., Pnueli, A.: From timed to hybrid systems. In: Real-Time: Theory and Practice, REX Workshop. pp. 447–484. Springer-Verlag (1992)
26. Martinez, J., Sañudo, I., Bertogna, M.: Analytical characterization of end-to-end communication delays with logical execution time. IEEE Tr. on Computer-Aided Design of Integrated Circuits and Systems **37**(11), 2244–2254 (November 2018). https://doi.org/10.1109/TCAD.2018.2857398
27. NASA Engineering and Safety Center: National highway traffic safety administration Toyota unintended acceleration investigation. Technical assessment report, NASA (January 18 2011)
28. Pree, W., Templ, J.: Modeling with the timing definition language (TDL). In: Broy, M., Krüger, I.H., Meisinger, M. (eds.) Automotive Software Workshop San Diego (ASWSD) on Model-Driven Development of Reliable Automotive Services. LNCS, vol. 4922, pp. 133–144. Springer (March 15-17 2006)
29. Resmerita, S., Naderlinger, A., Lukesch, S.: Efficient realization of logical execution times in legacy embedded software. In: ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE). pp. 36–45 (September 2017). https://doi.org/10.1145/3127041.3127054
30. Schneider, C., Spönemann, M., von Hanxleden, R.: Just model! – Putting automatic synthesis of node-link-diagrams into practice. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13). pp. 75–82. San Jose, CA, USA (Sep 2013)
31. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) **7**(3), 1–53 (2008)
32. Zhao, Y., Lee, E.A., Liu, J.: A programming model for time-synchronized distributed real-time systems. In: Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 259 – 268. IEEE (2007)