Deterministic Coordination across Multiple Timelines

MARTEN LOHSTROH, University of California, Berkeley, USA
SOROUSH BATENI, University of Texas at Dallas, USA
CHRISTIAN MENARD, TU Dresden, Germany
ALEXANDER SCHULZ-ROSENGARTEN, Kiel University, Germany
JERONIMO CASTRILLON, TU Dresden, Germany
EDWARD A. LEE, University of California, Berkeley, USA

We discuss a novel approach for constructing deterministic reactive systems that revolves around a temporal model that incorporates a multiplicity of timelines. This model is central to Lingua Franca (LF), a polyglot coordination language and compiler toolchain we are developing for the definition and composition of concurrent components called reactors, which are objects that react to and emit discrete events. Our temporal model differs from existing models like the logical execution time (LET) paradigm and synchronous languages in that it reflects that there are always at least two distinct timelines involved in a reactive system; a *logical* one and a *physical* one—and possibly multiple of each kind. This article explains how the relationship between events across timelines facilitates reasoning about consistency and availability across components in cyber-physical systems (CPSs).

CCS Concepts: • **Computing methodologies** → **Concurrent programming languages**; • **Software and its engineering** → **Distributed programming languages**; **Orchestration languages**; • **Computer systems organization** → *Real-time systems*; *Embedded and cyber-physical systems;*

Additional Key Words and Phrases: Time, coordination, concurrency, determinism

ACM Reference format:
Marten Lohstroh, Soroush Bateni, Christian Menard, Alexander Schulz-Rosengarten, Jeronimo Castrillon, and Edward A. Lee. 2024. Deterministic Coordination across Multiple Timelines. *ACM Trans. Embedd. Comput. Syst.* 23, 5, Article 77 (August 2024), 29 pages.
https://doi.org/10.1145/3615357

The work in this paper was supported in part by the National Science Foundation (NSF), awards #CNS-1836601 (Reconciling Safety with the Internet) and #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems) and the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by Denso, Siemens, and Toyota. This work was also supported in part by the German Federal Ministry of Education and Research through the project "E4C" (16ME0426K), the Software Campus program (01IS12051), and the "Souverän. Digital. Vernetzt" program in the joint project 6G-life (16KISK001K).
Authors' addresses: M. Lohstroh and E. A. Lee, University of California, Berkeley, 545Q Cory Hall, Berkeley, CA, 94720; e-mails: {marten, eal}@berkeley.edu; S. Bateni, University of Texas at Dallas, 800 W. Campbell Road, Richardson, TX, 75080; e-mail: soroush@utdallas.edu; C. Menard, TU Dresden, Chair for Compiler Construction, Georg-Schumann-Str. 11, 01069, Dresden, Germany; e-mail: christian.menard@tu-dresden.de; A. Schulz-Rosengarten, Kiel University, Department of Computer Science, Olshausenstr. 40, 24098, Kiel, Germany; e-mail: als@informatik.uni-kiel.de; J. Castrillon, TU Dresden, Chair for Compiler Construction, Georg-Schumann-Str. 7a, 01069, Dresden, Germany; e-mail: jeronimo.castrillon@tu-dresden.de.

1539-9087/2024/08-ART77
https://doi.org/10.1145/3615357

1 INTRODUCTION

Common software engineering approaches for expressing concurrent programs, such as threads [43], actors [2, 32], reactive programming [6], publish-subscribe systems [60], and even single-threaded event loops [3], make it difficult to achieve deterministic behavior. Nondeterministic behavior emergent in software built using the aforementioned programming models makes testing and verification efforts unnecessarily hard. Particularly for applications that are high in complexity and for which the cost of unintended system behavior is high, reproducibility is key. With the growing pervasiveness of networked computing and a trend toward integrating computationally demanding artificial intelligence components into real-time **cyber-physical systems (CPSs)** such as robots, autonomous vehicles, and so forth, the class of highly concurrent safety-critical systems is rapidly expanding. We believe that a deterministic programming model organized around a sophisticated notion of time can help meet the demands of the next generation of cyber-physical systems.

It has been shown previously that even for applications that do not have real-time requirements, a semantic notion of time and the use of measurements of the passing of physical time can be powerful tools for achieving a measure of consistency in concurrent and distributed software [42, 45, 71]. Google's Cloud Spanner [16], for example, uses timestamps derived from physical clocks to define the behavior of a distributed database system; Spanner provides an existence proof that this technique works at scale. Moreover, logical time, as used in synchronous languages [7], for example, can provide a foundation for a deterministic semantics in concurrent programs.

Contributions. We propose a coordination model that involves a multiplicity of distinct logical and physical timelines that are used to determine in which order events are observed and whether or not they are handled before a specified deadline. We expose this model in **Lingua Franca (LF)**, a polyglot coordination language that we designed to augment mainstream programming languages with a coordination layer based on a discrete event semantics. Our language incorporates verbatim target-language code, allowing LF programs to benefit from the vast number of libraries and advanced compilers and interpreters of established programming languages.

Outline. The article is organized as follows. Section 2 gives a motivating example showing how existing coordination paradigms fall short of delivering repeatable and testable concurrent behavior. Section 3 discusses the fundamental challenges and opportunities of using time for specifying concurrent behavior. Section 4 introduces LF and explains how it allows a programmer to relate events across distinct timelines. Section 5 gives a series of illustrative examples and uses these examples to compare the LF semantics to the logical execution time principle. Section 6 shows how the discrete event semantics of LF can be preserved across distributed components. Finally, Section 7 discusses related work, and Section 8 concludes.

2 MOTIVATION

Consider the following simple but challenging problem. Suppose that a commercial aircraft manufacturer wishes to automate the opening of an aircraft door. Consider a networked software component residing in the door that provides two services, `open` and `disarm`. The `disarm` service disables deployment of emergency escape slides if the door is armed, and the `open` service opens the door. If the door is opened when it is armed, then the slides will deploy. The challenge problem is to decide what the software should do when it receives an `open` command from the network.

Of course, the software could simply open the door, but this is dangerous without additional guarantees from the environment. Network delays or nondeterminism may result in out-of-order message arrival, potentially causing a `disarm` message that was sent prior to the `open` command

```
1  actor Door {
2      closed = true;
3      armed = true;
4      handler disarm(){
5          // ... actuate ...
6          armed = false;
7      }
8      handler open(arg){
9          // ... actuate ...
10         closed = false;
11     }
12 }
```

```
13 actor Cockpit {
14     handler main {
15         d = new Door();
16         d.disarm();
17         d.open();
18     }
19 }
```

Fig. 1. Pseudo-code for an actor network that is deterministic under reasonable assumptions about message passing.

to be received *after* the open command. In that case, simply opening the door would lead to an unintended emergency slide deployment.

A number of reasonably disciplined techniques have evolved to coordinate distributed programs, including publish-and-subscribe, actors, service-oriented architectures, and distributed shared memory. None of these, however, provides enough control over ordering to resolve this simple problem satisfactorily.

Consider actors [2, 32], as realized in Erlang [4], Akka [68], and Ray [62]. The pseudo-code example given in Figure 1 illustrates an actor-based solution. The actor `Cockpit` sends two messages, `disarm` and `open`, to the actor `Door`. Although many actor languages make the sending of messages appear like remote procedure calls, their semantics is "send and forget," a feature that enables parallel and distributed execution but poses challenges to coordination. Without further assumptions or explicit synchronization, there is no guarantee that the `Door` actor processes `disarm` before `open`.

Under mild assumptions about the network (i.e., reliable in-order message delivery, which TCP can provide), the program in Figure 1, subject to the constraint that handlers are mutually exclusive, is deterministic [52]. However, this property breaks with even the slightest change to the actor network. Consider the minor elaboration shown in Figure 2. This program has a third actor, `Relay`, that simply passes the `disarm` message from `Cockpit` on to `Door`.[1] This innocent change has troubling consequences. The execution is no longer deterministic under any reasonable assumptions about the network, which could cause an unintended deployment of the emergency slides. This type of nondeterminism is endemic to the Hewitt actor model. Moreover, it is difficult to change the program in Figure 2 to consistently behave correctly [52].

In robotic systems, such as in ROS [66], or in the Internet of Things, such as in MQTT [34], publish-and-subscribe protocols are widely used to coordinate software components. Since such communication fabrics provide no assurances about the order of message delivery or the order of message handling, they are prone to the same problem of nondeterministic behavior. ROS 2 uses **Data Distribution Service (DDS)** [64], which supports priorities on messages. This might seem like a solution to the problem, but priorities are not a semantic property. They are a quality-of-service property rather than a correctness criterion. Hence, they could even *mask* a semantic problem in a design, making it less likely to show up in testing. This will also make it less likely to show up in the field, but even the rare occurrence of a dangerous and life-threatening action is problematic.

[1] In a real application, instead of just relaying the message, the actor could interrogate sensors to determine that a passenger boarding ramp has been placed outside the door before relaying the `disarm` message.

```
actor Cockpit {
  handler main {
    d = new Door();
    r = new Relay();
    r.rly(d);
    d.open();
  }
}
```

```
actor Relay {
  handler rly (x){
    x.disarm();
  }
}
```

Fig. 2. Modification of the code in Figure 1 yielding a nondeterministic program. The actor Door remains the same.

Another approach could rely on a distributed shared memory architecture, often realized using the tuple space concept of Linda [25]. However, a shared memory model provides even less support to prevent the sorts of problems we highlight here.

Service-oriented architectures, widely used for web applications (e.g., Apache Thrift [1]), are increasingly applied in cyber-physical systems (e.g., the AUTOSAR Adaptive Platform [5]). But they too admit nondeterminism. Recent work shows that this nondeterminism can have fatal consequences in safety-critical applications [61]. Menard et al. [61] leverage the same underlying principles that the work in this article builds upon, but they only furnish basic runtime support that allows programmers to manually construct the kind of "glue code" that the LF compiler produces automatically.

The approach we advocate in this article will prove extremely simple, as it should be for such a simple problem. We add timestamps to every sent message, and, upon receiving a timestamped message, the Door waits until its physical clock hits a precomputed threshold before processing the message. The threshold ensures, under clearly stated assumptions, that all messages are handled in timestamp order. The question remains: how long should the Door have to wait?

3 REASONING ABOUT TIME

3.1 Observation and Relativity

It is impossible, from first principles in physics, to determine the order in which two geographically separated events occur. There is no such thing in physics as the "true" order in which separated events occur. There is only the order seen by an observer, and two observers may see different orders. Hence, it would be an unrealistic goal to require that if a `disarm` message is "truly" sent before an `open` message, then the door will be disarmed before it is opened. To use such a requirement, we would have to identify the observer that determines the outcome of the predicate "before."

One choice of observer, of course, is the receiver of the messages, the microprocessor in the door that performs the `disarm` and `open` services. This is the choice made in an actor model (as well as publish-and-subscribe and service-oriented models), but as we have shown, it leads to clearly undesirable outcomes. Even if the `disarm` and `open` messages originate from the same source, they may arrive out of order. The originator sees a different order from the recipient, as shown in Figure 3.

Only if instead of relying on a physical notion of time we define a *logical* or *semantic* notion of time does it become possible to ensure that every observer sees events in the same order. This will require a careful definition of "time" as a semantic property of programs. We will also have to stop pretending that our logical notion of time *is* physical time and instead accept a multiplicity of observers and understand the relationships between their timelines.

One way to provide a semantic notion of time is to use numerical timestamps [41]. If messages carry timestamps, then our requirement can be that every actor processes messages in timestamp

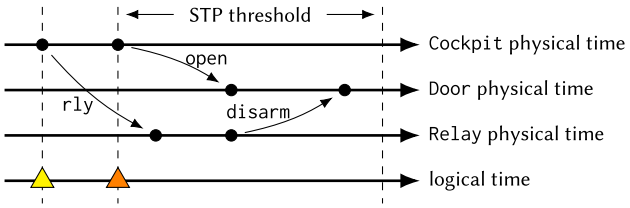


Fig. 3. Different observers may see events in a different order. An additional logical timeline allows to establish a global ordering. After a certain safe-to-process (STP) threshold, Door received all relevant messages and can use the logical timeline to determine that disarm should be processed *before* open.

order. If we further require that messages with identical timestamps be processed in a predefined deterministic order, then our semantics will ensure that any two actors with access to the same messages will agree on their order. We know from experience with distributed discrete-event simulators, however, that it is challenging in a distributed system to preserve timestamp order [23]. Moreover, here, we are not interested in *simulation*. We are interested in cyber-physical *execution*, where physical time and (imperfect) measurements of physical time play an important role. The methods used for distributed simulation will have to be adapted, as we do here.

The use of timestamps superimposes on our distributed system a logical timeline that must coexist with a multiplicity of timelines, measurements of physical time, and actual physical time. We will show how physical clocks can be used to create logical timestamps and how the relationship between timestamps and physical clocks can lend a rigorous meaning to deadlines. Moreover, we give a mechanism that, once the timestamps of messages have been determined, is deterministic, under clearly stated assumptions. When the assumptions are met, the system behavior is repeatable, in that, given the same timestamped inputs, the response will always be the same. This determinism also makes systems more testable. A set of timestamped inputs forms a test vector, and the system defines the one and only correct response to this test vector. Moreover, violations of the assumptions are detectable at runtime. When any component sees messages out of timestamp order, one of the assumptions has been violated. This detectability enables the design of fault-tolerant systems.

In the aircraft door example, when the Door component receives a timestamped open message, it waits until its local physical clock hits a precomputed threshold before acting on that message (cf. Figure 3). This guarantees that the open message will be handled in timestamp order relative to other messages, including any disarm messages that may originate anywhere in the system. The assumptions will include a bound E on the clock synchronization error, a bound L on the network latency, and a bound X on the execution time of certain pieces of code. What bounds are acceptable is application dependent. Existing technologies allow to tighten bounds on E [36], L [39], and X [70, 72].

In reality, *any* reasonable handling of an open message has to make these same assumptions. If there really is no bound on network latency, how can we possibly reason about the order in which messages are handled? If clocks differ wildly across a distributed system, how can we expect any coherent notion of "before"? To address this reality, we created the coordination language LF, in which these assumptions are explicit and quantified and their violation detectable.

3.2 Logical and Physical Time

The logical ordering of events in software gives rise to a logical timeline, which is different from the timeline that tracks side effects (such as the blinking of an LED) observed in the physical environment. In a CPS, both timelines matter, but we must be careful not to conflate the two.

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 $\mathbb{I}$.

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. For example, $\mathbb{T}$ and $\mathbb{I}$ could both be the set of 64-bit integers, where a $T \in \mathbb{T}$ is a POSIX-compliant representation of time representing the number of nanoseconds that have elapsed since midnight, January 1, 1970, Greenwich mean time. The largest and smallest 64-bit integers could represent ∞ and $-\infty$, respectively, where 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. Such overflow could saturate at ∞ or $-\infty$, and as a consequence, addition will no longer be 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 $\mathbb{G}$. The term "tag" is inspired by the tagged signal model [48]. 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 [15, 17, 55]. We use the term *tag* rather than timestamp to allow for such a richer model of logical time. For the purposes of this article, 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 lowercase $g \in \mathbb{G}$ and measurements of physical time $T \in \mathbb{T}$ with uppercase.

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 LF, for any tag $g = (t, m) \in \mathbb{G}$, $\mathcal{T}(g) = t$. Hence, retrieving a timestamp from a tag simply ignores 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 LF, 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 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.

4 LINGUA FRANCA

The focus of LF is on network-integrated reactive and cyber-physical systems [44]. Many computing activities can be viewed through that lens. They all have in common that they benefit from repeatability and testability, meaning that their behavior in response to some specified external stimulus is well defined and consistent across operating conditions. Many such systems are also time sensitive and safety critical. Our model of time furnishes a well-defined semantics of the interaction between reactive software components and physical processes, and allows timing constraints to be specified in the software.

The LF programming paradigm is based on the reactor model [50, 51], in which *reactors* are components that maintain state and can contain other reactors. Reactors carry functionality inside of *reactions*. Reactions bear some resemblance to object-oriented methods, but rather than being invoked directly, they are triggered by the occurrence of an event on a port or action. While *ports* relay events between reactors via *connections*, *actions* relay events internal to the reactor. Events have a tag and can carry a value of some datatype. Like signals in Esterel [8], a port or action has at most one event at a given tag. Reactions are logically instantaneous, meaning that logical time does not advance during their execution and that any outputs produced on ports by a reaction have the same tag as the triggering event (this contrasts with the logical execution time paradigm, as explained in Section 5). Scheduling an action, unlike producing an output on an output port, yields an event with a tag strictly greater than the current logical time. Consequently, ports are a mechanism to (logically) synchronously communicate across reactors, while actions are a mechanism to advance logical time.

Reactions can access (and be triggered by) ports of their own reactor, and also ports of reactors that are directly contained within that reactor. Importantly, scoping rules require each port that

a reaction references to be declared explicitly in the signature of the reaction. The interfaces of reactions, therefore, along with the explicit connections between ports, contain all information necessary to devise a concurrent execution policy that observes all data dependencies in a reactor program—*without the need for any static analysis of the reaction code*, which is written in a target language. This feature is the key enabler of the polyglot nature of LF.

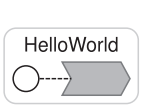
It is important to note that the execution of a reaction is guaranteed to be mutually exclusive and ordered deterministically with respect to other reactions in the *same* reactor (not with respect to reactions in other reactors, not even contained ones). Should any two reactions of the same reactor trigger at the same tag, their execution follows the order in which the reactions are defined. The reason for this is to avoid any race conditions on ports and shared state that reactions may access.

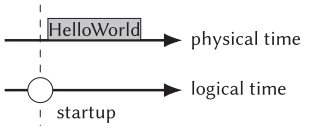
Reactors are causal discrete-event components, where "causal" can be formalized as a strictly contracting function on a generalized metric space, and a constructive version of the Banach fixed-point theorem [15, 58, 59] can be used to prove determinism of a composition of reactors that are free of causality loops (see Section 4.3). This gives a denotational semantics to reactors [54]. A full treatment of the formal aspects of reactors, however, is outside of the scope of this article.

4.1 Hello World

A minimal LF program printing "Hello World!" can be given as follows:

```
1 target C;
2
3 main reactor HelloWorld {
4     reaction(startup) {=
5         printf("Hello World!\n");
6     =}
7 }
```





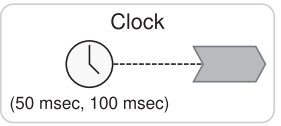
Each LF program starts with a *target declaration* that specifies the target language and may optionally contain further configuration. Currently, LF supports C, C++, Python, TypeScript, and Rust [22] as target languages. Each LF program further defines a main reactor. Analogous to the main function in C/C++, this serves as the entry point to the program's execution. The `HelloWorld` reactor in the example above defines a single reaction. This reaction reacts to the built-in `startup` action, which is triggered once when the program begins executing. In a federated LF program (see Section 6), all components start executing at the same logical time, and hence, all `startup` triggers across the system are logically simultaneous. The reaction's functionality, printing the string "Hello World", is given within the {= ··· =} delimiters in target code (C in this case). This quotation mechanism allows embedding arbitrary target code in LF programs. The LF compiler does not analyze or parse the target code and instead relies on the target language compiler to perform language-specific checks.

4.2 Expressing Timed Behavior

To express timed behavior, reactions can also be scheduled to occur at some particular future time instant. These can be one-shot reactions, periodic reactions, reactions that are offset by an arbitrary logical delay, or reactions in response to an external stimulus such as an interrupt or asynchronous callback.

4.2.1 Timers. In LF *timers* are used to specify one-shot or periodic triggers. A periodic timer can be given as follows:

```
1 target C
2 main reactor Clock {
3     timer t(50 ms, 100 ms)
4     output y:int
5     reaction(t) -> y {=
6         lf_set(y, 42);
7     =}
8 }
```



The timer is named `t`, and its first triggering is 50 ms after the (logical) start of execution of the program. Subsequent triggers occur every 100 ms. If the second argument on line 3 is omitted, then the trigger occurs only once. If both arguments are omitted, then the timer is equivalent to `startup`. If any other timer anywhere in the program triggers events at the same logical time as this timer, then those events are logically simultaneous. Note that these times are *logical times* that will be aligned on a best-effort basis with physical time, and the accuracy of such alignment will depend on the real-time capabilities of the execution platform. But the order in which events are seen will not depend on these real-time capabilities. The order is defined by the tags (see Section 3.2). In LF, we assume that logical time always lags behind physical time. This practically means that no event will be handled until physical time is greater than the tag of the event.

In the above LF program, a reaction is defined that is periodically triggered by the timer `t`. The signature for this reaction, line 5, further indicates that the reaction (possibly) produces an output on the port named `y`. The reaction code uses the library function `set` to set the value of the output port at the logical time at which the reaction triggers. Hence, the above reactor will produce the output value 42 at logical times 50 ms, 150 ms, 250 ms, and so forth after the logical start time of the program.

4.2.2 Actions. Like a timer, an action triggers reactions, but instead of occurring at fixed, predefined times, actions can be less regular. An action is scheduled when the target code calls the `lf_schedule` function, which takes two arguments, an action and a time offset. Consider the reactor shown in Figure 4. This reactor produces its first output 100 ms after startup and then produces outputs with intervals that increase by 100 ms each time. To accomplish this behavior the reactor defines on line 3 an action named `a` with a *minimum* delay of 100 ms. At startup, on line 6, the first reaction calls `lf_schedule`, passing it the action `a`, and an *additional* delay of zero.

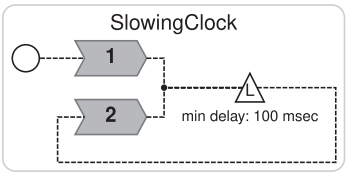
The second reaction (lines 8 to 15) is triggered by the action `a` and prints the elapsed logical time since execution start. It then calls `lf_schedule` again, this time using the state variable named `interval` to specify an additional delay. It then increments the interval by 100 ms.

4.2.3 Logical vs. Physical Actions. The action `a` in the previous example is a *logical* action, which means that when a reaction calls `lf_schedule`, the tag assigned to the resulting event depends on the current logical time t. The new tag assigned to `a` is calculated as $t + d_1 + d_2$, where d_1 is the extra delay passed to `lf_schedule` and d_2 is the minimum delay given in the action declaration.[2] Since logical time does not elapse during reaction execution, the printed outputs are deterministic.

A logical action can only be scheduled in a reaction, and therefore can only create future events in immediate response to earlier events. Suppose that we wish instead to create an event in response to something external, such as an interrupt occurring or callback function being called.

[2]The reason for a minimal delay specified separately from the extra delay passed to the `lf_schedule` function is that there is useful static analysis that can depend on this number, for example, to determine schedulability (beyond the scope of this article). The minimum delay is visible without parsing and analyzing target code.

```
target C
main reactor SlowingClock {
    logical action a(100 ms)
    state interval:time = 100 ms
    reaction(startup) -> a {=
        lf_schedule(a, 0);
    =}
    reaction(a) -> a {=
        printf("Logical time since start: \
            \%lld nsec.\n",
            get_elapsed_logical_time()
        );
        lf_schedule(a, self->interval);
        self->interval += MS(100);
    =}
}
```




```
Logical time since start: 100000000 nsec.
Logical time since start: 300000000 nsec.
Logical time since start: 600000000 nsec.
Logical time since start: 1000000000 nsec.
...
```

Fig. 4. Example use of a logical action.

Examples of such an event would be user input, an interrupt-driven sensor, or network messages coming from outside the (distributed) LF program. For this purpose, LF provides physical actions.

A physical action is declared as follows:

```
physical action a:type;
```

The `lf_schedule` function is invoked *outside* of a reaction, asynchronously, during or between executions of reactions. To ensure that the tag assigned to the scheduled event is strictly larger than that of any event that the reactor has reacted to (or is reacting to), LF ensures that this reactor's logical time never gets ahead of physical time as reported by the physical clock on the execution platform. The fact that logical time always lags behind physical time accommodates the production of sporadic events in the environment. It ensures that assigning such events a tag based on a current reading of the physical clock is always *safe* in the sense that doing so will not give rise to the possibility of events being handled out of order. Once an external event has been assigned a tag, the order of its processing is determined exclusively by its tag.

4.3 Composition

Reactors can be composed by drawing connections between their ports. A port may be used as an *effect* by some reaction in an upstream reactor that can produce events on the port, and used as a *trigger* or *source* by a reaction in a downstream reactor that can respond to events on that port. A connection therefore may imply a dependency between two reactions. Unless the connection has a logical delay (see Section 4.5), the dependency is logically instantaneous in that it implies that the two reactions may execute at the same tag, but must execute in order. An additional ordering constraint is that any two reactions of the same reactor that execute at the same tag must execute in the order of their declarations (to preserve determinism). Together, these ordering constraints form a graph where the nodes are reactions and the edges are dependencies. This graph is required to be acyclic. Cycles in the graph are *causality loops* that render the program unschedulable. Causality loops can be avoided by reordering reactions in a reactor or by introducing logical delays on connections, but this cannot be done automatically because it changes the program logic. The compiler provides suggested changes to make the graph acyclic.

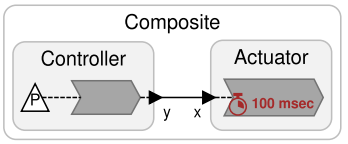
Causality loops should not be confused with "stuttering" Zeno conditions [49], which occur when subsequent events iterate in superdense time, not increasing the time value of their tag but

only the microstep index. In such situation, the system will stop being responsive to its physical environment. But not all programs are meant to have such interactions. For purely computational programs that do not need to maintain responsiveness while computing, stuttering Zeno behavior is not a problem. Apart from the fact that the scheduling of events happens inside of reactions, which the compiler does not inspect, it is generally undecidable whether a program exhibits Zeno behavior [54]. Hence, it is considered the programmer's responsibility to avoid Zeno behavior if necessary, just like it is the programmer's responsibility to not write infinite while loops.

4.4 Deadlines

LF includes a notion of a *deadline*, which is a relation between logical time and physical time. Specifically, a program may specify that the invocation of the reactions to some event must occur within some physical-time interval measured from the logical time of the event. If a deadline is violated, then instead of allowing the tardy event to trigger the reaction, the code in the body of the attached deadline miss handler is executed. For example:

```
1  reactor Controller {
2      physical action sensor:int
3      output y:int
4      // ...
5      reaction(sensor) -> y {=
6          int control = calculate(sensor_value);
7          lf_set(y, control);
8      =}
9  }
10 reactor Actuator {
11     input x:int
12     reaction(x) {=
13         // Time-sensitive code
14     =} deadline(100 ms) {=
15         printf("*** Deadline miss!\n");
16     =}
17 }
18 main reactor Composite {
19     c = new Controller();
20     a = new Actuator();
21     c.y -> a.x;
22 }
```



The above program illustrates how the end-to-end latency between a sensor and an actuator can be bounded by a deadline. The program instantiates two reactors `c` and `a`, instances of `Controller` and `Actuator`, respectively. The physical action `sensor` on line 2 will be triggered by an asynchronous call to the `lf_schedule` function, for example, within an **interrupt service routine (ISR)** handling the sensor (that code is not shown). The action will be assigned a tag based on what the physical clock indicates when the ISR is invoked. That tag, therefore, is a measure of the physical time at which the sensor triggered. The reaction to `sensor`, on line 6, performs some calculation and sends a control message to its output port. Line 21 connects that output to the input `x` of the actuator.

The actuator's reaction to the input `x` declares a deadline of 100 ms on line 14 followed by a deadline violation handler. If this reaction is not invoked within 100 ms of the tag of the input, as measured by the local physical clock, then rather than executing the time-sensitive code in the reaction, the deadline violation is handled. The deadline, therefore, is expressing a requirement

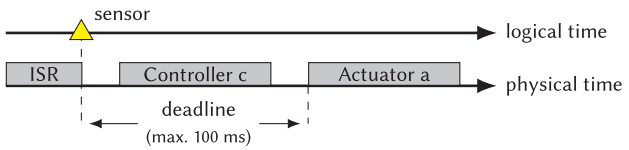


Fig. 5. A deadline defines the maximum delay between the logical time of an event and the physical time of the start of a reaction that it triggers.

that the calculation on line 6 (plus any overhead) not take more than 100 ms (in physical time). This relation across timelines is illustrated in Figure 5.

The presence of such deadlines in the LF code enables the code generator to synthesize earliest-deadline-first scheduling policies. Dependencies between reactions are known statically through the enforcement of scoping rules in the language. A reaction can only produce outputs on ports that it declares among its effects; e.g., a reaction with the signature `reaction(a) b -> c` is triggered by `a`, may read `b`, and may set `c`. If it sets `d`, a compile error results. In other words, the reaction signatures combined with the connections between reactors provide a full view into the dependencies of the program. This makes it easy to find all reactions upstream of a deadline and prioritize them accordingly. Of course, depending on the target language, the scoping rules can be evaded by establishing "under the table" communication through the use of pointers or shared variables. However, the point of LF is not to make it impossible to subvert the model of computation, but to make it unlikely that a programmer would do this by accident.

Note that the deadline construct in LF admits nondeterminism. The program will be deterministic only if the deadlines are not violated. Whether the deadline is violated or not depends on factors outside the semantics of LF. Deadline reactions in LF, therefore, should be thought of as *fault handlers*, and deadline specifications as requirements. When a requirement is violated, a fault handler is invoked.

4.5 Logical Time Delays

A logical time delay between two reactions can be implemented using a logical action. As a convenience, LF allows for connections to be annotated with an `after`-clause that specifies a time delay. Such delay effectively shifts a produced output along the logical timeline. As such, this mechanism can be used to reduce the amount by which logical time lags physical time and account for the execution time of reactions. By choosing the delay between two reactions connected to one another via ports—a producer and a consumer—such that the delay exceeds the **worst-case execution time (WCET)** of the producer, the tags of the events are always greater than the physical time at which they are produced. This effectively assigns a logical execution time [29] to the producer, allowing the execution of the consumer to be timed more precisely with respect to physical time.

5 LET AND MORE IN LINGUA FRANCA

The **logical execution time (LET)** principle, introduced by Henzinger et al. [29], bears a strong enough resemblance to the model of computation in Lingua Franca that it deserves a detailed comparison. The short story is that LF programs can realize LET, but LF is more general and can realize patterns not easily supported by any LET framework that the authors know of. The Giotto programming language [31] elegantly realizes the LET principle in the form of a coordination language, where the business logic of programs is realized in a conventional language (such as C), but the modal behavior, concurrency, and timing are orchestrated by a runtime engine that closely follows the LET principle. As it is also a coordination language, LF resembles Giotto.

The early LET work inspired quite a bit follow-up work, including applications to distributed real-time automotive software [26] and automotive multicore software [10]. The LET principle has also been applied to programming time-predictable multicore processors [37], has been used to facilitate parallel execution of legacy software on multicore [67], and has been leveraged for schedulability analysis [33]. Whereas in Giotto execution of components is time driven, the language extensions in xGiotto support asynchronous events [28], thereby bringing Giotto a bit closer to Lingua Franca. **System-level LET (SL-LET)** [26, 38] is a recent extension of the LET principle to distributed systems, where communication, like computation, takes a specified amount of logical time. In federated LF (see Section 6), the same effect is accomplished with logical delays on communication paths between federates.

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 pre-specified 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 as 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 [7] are based on a hypothesis of zero execution time. Lingua Franca effectively combines these two alternatives, enabling zero and non-zero logical execution times, and also, like xGiotto and unlike Giotto, support non-periodic, event-triggered actions.

In this article, 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. Lingua Franca generalizes LET to preserve determinism while reducing 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. Hence, in LF, the physical and logical timelines are more distinct.

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 LF distinguishes logical time from physical time and enables alignment at cyber-physical interaction points [53]. As we go through the examples, we explain in more detail how execution of LF programs works.

5.1 Periodic Polled Control System

Consider a cascade composition of tasks between a sensor and an actuator as shown in Figure 6. One might find this example in the software portion of a feedback control system. The figure on the bottom is automatically generated by the Lingua Franca IDE called Epoch.[3,4] The chevrons

[3]Epoch is available for download at https://releases.lf-lang.org

[4]The diagram synthesis feature was created by one of the authors (Schulz-Rosengarten) using the graphical layout tools from the KIELER Lightweight Diagrams framework [69] (see https://rtsys.informatik.uni-kiel.de/kieler).

```
target C;
reactor Sensor(p:time(10 msec)) {
    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 msec)) {
    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;
}
```

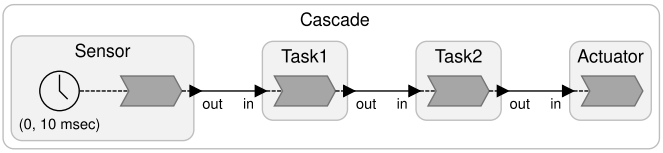


Fig. 6. Cascade composition.

in the figure represent reactions, and their dependencies on inputs and their ability to produce outputs are 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, 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, $i = 0, 1, 2, \ldots$, 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. After all, our semantics require that all events be handled in tag order.

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.

In Figure 7 we modify the last two lines of the program in Figure 6, 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.

First, Task1 and Task2 can now execute in a pipelined fashion, in parallel, exploiting multiple cores if they are available. 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}.

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

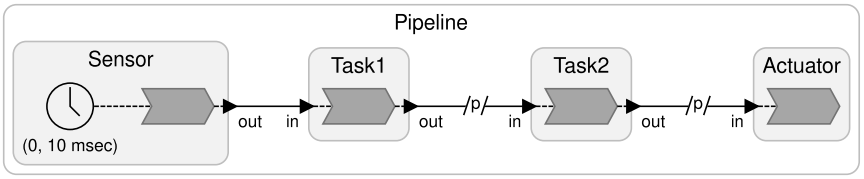


Fig. 7. Pipeline with logical delays, emulating LET.

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 Figure 6, the data delivered to the Actuator is based on older sensor input, so the designer is faced with a tradeoff between data age [13] and predictable, repeatable timing. In many safety-critical systems, repeatable timing is extremely valuable; for one, it greatly enhances the value of testing [46, 57].

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 of 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 Figure 8. The LF 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.

In addition, the deadline construct provides a *fault handling* mechanism. Line 6 in Figure 8 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 LF are still logically instantaneous even if their outputs are subjected to a logical delay using the `after` keyword. In LF, an input or output port is modeled as a function $P\colon \mathbb{G} \rightarrow V \cup \{\epsilon\}$, 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.

```
1  // ...
2  reactor Actuator {
3      input in:int
4      reaction(in) {=
5          // ... drive actuator ...
6      =} deadline (1 ms) {=
7          // ... handle a deadline miss ...
8      =}
9  }
10 main reactor(p:time(10 ms)) {
11     // ...
12     t1.out -> t2.in after p
13     t2.out -> a.in after p
14 }
```

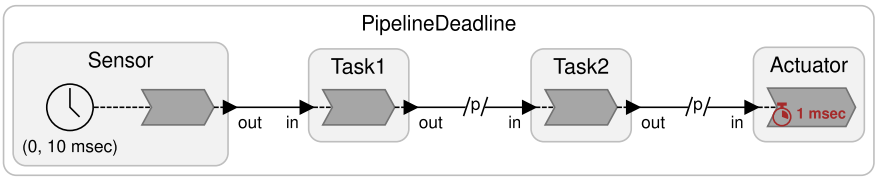


Fig. 8. Pipeline with deadline.

5.2 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 Figure 9. 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 `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 `schedule()`, the LF 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 Figure 7 will *not* yield parallel execution even if it does emulate LET and help to regulate the timing of actuation relative to sensing. 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. However, if the physical action is used in a federated program (see Section 6), then parallel execution *is* possible because Task2 may be still processing the previous event when a new event arrives.

An unconstrained physical action like that of Figure 9 runs a risk of overwhelming the software system and disrupting timing. If `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

```
// ...
reactor Event {
    output out:int
    physical action a
    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
}
```

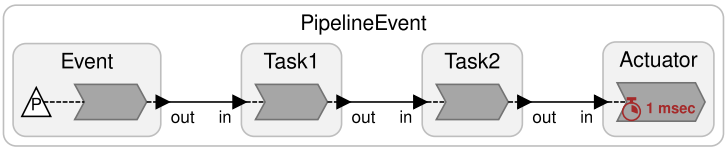


Fig. 9. Event-triggered pipeline.

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 Figure 10. 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.

5.3 Merging Events with Periodic Tasks

In cyber-physical systems, it is common for software to handle combinations of events originating from the physical environment and periodic events that originate from the software itself. For example, a feedback control system may operate with a regular sample rate, but sporadic events may result in changes in the control laws. Such sporadic events are called "asynchronous" because they happen asynchronously with respect to the current logical time implied by the tag of the last handled event. As described in Section 4.2.3, such asynchronous events enter the system through

```
// ...
reactor Event {
    output out:int
    physical action a(0, 10 ms)
    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 after 10 ms
    t2.out -> a.in after 10 ms
}
```

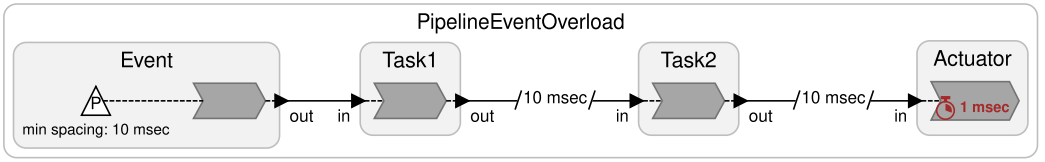


Fig. 10. Event-triggered pipeline optimized for overload conditions.

physical actions with a tag based on a reading from the physical clock. Yet it is possible to still achieve regular, tightly controlled timing in the face of such events.

Figure 11 shows an example where a *logical action* 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. That call to `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.

5.4 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. Figure 12 shows a variant of Figure 11 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 the reactor model, if the two input ports have simultaneous input events, then the first reaction executes to

```
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.
        lf_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 = current_time % MSEC(10) + 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 ...
        }
    =}
}
...
```

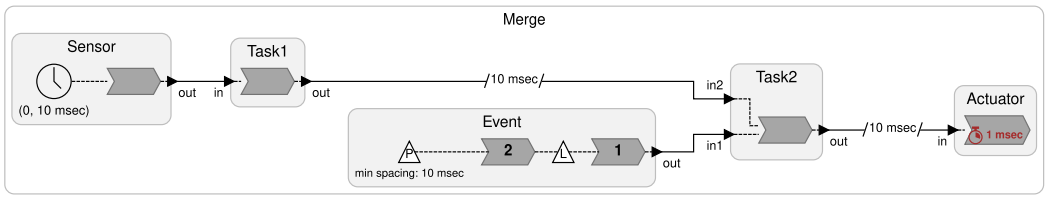


Fig. 11. Merging of asynchronous events with periodic ones.

completion before the second reaction executes, so access to the state variable is mutually exclusive and deterministically ordered. No such ordering is enforced between reactions across different reactors, enabling parallel execution of logically simultaneously triggered reactions that do not share state.

Notice in Figure 12 that we no longer need the logical action of Figure 11. The effect of the new control law is guaranteed to align with the 10 ms timing of the periodic events.

6 FEDERATED REACTORS

LF programs can also be *federated*. An ordinary LF program can be turned into a federated one simply by replacing the `main` modifier of the top-level reactor with the `federated` keyword. In a federated reactor, each reactor contained in it will become a distinct process, called a "federate," that can be mapped to a host, giving rise to a distributed system. The semantics of LF is preserved

```
// ...
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 {=
        lf_set(out, a->value);
    =}
}
// ...
```

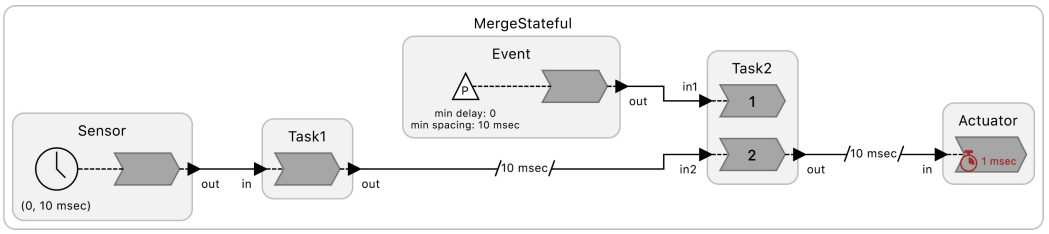


Fig. 12. Reactor with a state variable.

across federates by the addition of a coordination layer that controls the semantics of communication between federates. This coordination layer can either be *centralized*, meaning that a central coordinator (called the **Runtime Infrastructure (RTI)**) is in charge of the coordination, or it can be *decentralized*, meaning that the coordination is done at the federate level based on certain assumptions about the network delay, execution overheads, and physical clocks.

6.1 Federated Aircraft Door

A federated reactor that implements the aircraft door example is depicted in Figure 13. If a reactor has multiple reactions, the reactions are labeled with numbers that indicate the order in which logically simultaneous reactions will be executed.

Let us look at the `Cockpit` implementation first. Reaction 1 of the `Cockpit` reactor is triggered by the startup action (denoted by a circle). It carries out initialization, involving setting up callbacks or interrupt service routines that will be called in response to a signal coming from physical buttons. The pressing of a button will result in the scheduling of a physical action (denoted by a triangle labeled "P") that triggers either of the other two reactions, each of which sets the value of their respective output ports.

The `Relay` reactor performs further checks in order to determine whether it is safe to disarm the emergency escape slides. It interrogates sensors to verify that a passenger boarding ramp has been placed outside the door. Only if that is the case does it forward the `disarm` message to `Door`.

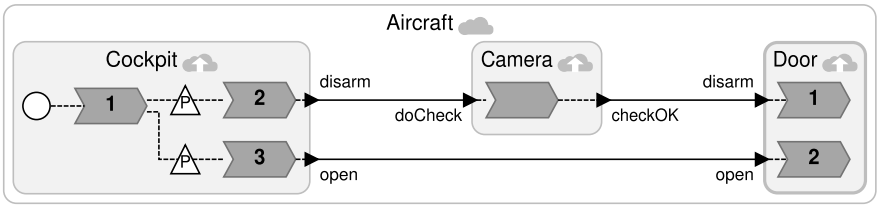


Fig. 13. A federated LF program implementing the aircraft door example. Each reactor may run on a different host machine.

The `Door` reactor simply responds to events on its `disarm` and `open` ports, as one would expect, by actuating the door. Note that our ordering of reactions within the `Door` reactor ensures that if the cockpit sends an `open` and `disarm` message at the same logical time (i.e., bearing the same tag), we guarantee that the door will be disarmed before it is opened.

6.2 Coordinating Federated Execution

In the LF compiler, regular reactors are turned into federates by substituting their connections to other federates with reactions that send and receive messages over the network. By default, LF ensures that event tags are preserved between federates using coordination techniques that will be described in this section. However, this default behavior can be overridden if there is no need to preserve tags by using "physical connections," using the `~>` operator instead of `->` to make connections between federates. On a physical connection, the logical time of any event at the receiver will be set to the physical time at which the event is received. This allows reactors to express the nondeterministic behavior of actors where this is applicable.

To ensure determinism in a federated program, however, it is essential to preserve tags across networked communication. For this, it is necessary to transmit tags along with the messages. A more subtle issue is that a federate must avoid advancing logical time ahead of the tags of messages it has not yet seen. This problem has many possible solutions, many of them realized in simulation tools [23]. However, LF is not a simulation but an implementation language, which introduces unique problems.

The centralized coordination in LF (which, as mentioned before, uses a centralized controller called the RTI) is similar to several tools that implement the **High Level Architecture (HLA)** standard [40]. In this approach, each federate has two key responsibilities. It must consult with the RTI before advancing logical time, and it must inform the RTI of the earliest logical time at which it may send a message over the network. The RTI will in turn grant advancements of logical time to federates in a manner that ensures all events are processed in tag order. This centralized approach to coordination, however, has three key disadvantages. First, the RTI can become a bottleneck for performance since all messages must flow through it. Second, the RTI is a single point of failure. Third, if a physical action can trigger an outgoing network message, then the earliest next event time is never larger than the time of the physical clock. This can lead to slow advancement of logical time with many messages exchanged with the RTI.

The decentralized coordination in LF is an extension of a technique called PTIDES [71], which has none of the aforementioned disadvantages. PTIDES, however, requires that the physical clocks on all federates be synchronized with some bounded error, using, for example, NTP or IEEE 1588 [21]. This requirement is relaxed in LF by providing a software-level clock synchronization based on HUYGENS [27]. PTIDES also requires being able to bound network latencies and (certain) execution times. These three bounds (clock synchronization error, network latencies, and certain

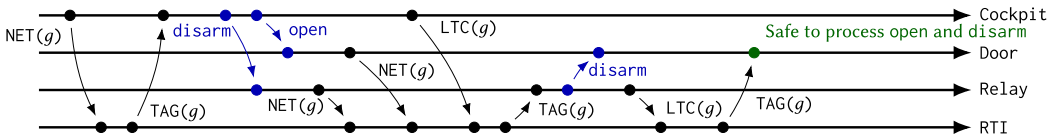


Fig. 14. Network messages exchanged between federates and the RTI for the aircraft door example, mapped over their respective physical timelines. Note that `disarm` and `open` messages (marked blue) are routed through the RTI, but this detail is omitted for simplicity.

execution times) have to be made explicit. The technique used by PTIDES has been shown to scale to very large systems; it is used in Google Spanner, a global database system that coordinates thousands of servers [16].

Like the deadline handler, Lingua Franca provides a fault handling mechanism when network latencies, clock synchronization error, or execution times violate the assumptions the programmer has made. As long as these (explicit) assumptions are satisfied, the overall distributed program remains deterministic. When the assumptions are violated, the fault handlers will be invoked instead of the regular reactions, so designers can build application-specific ways of dealing with networking problems.

Brewer's well-known CAP theorem [14] asserts that when network delays increase (the "P" in CAP refers to Partitioning), one must sacrifice either availability (the "A" in CAP) or consistency (the "C"). In a prior work, we have shown that the centralized control of federated Lingua Franca programs preserves consistency at the expense of availability when network delays get large, where the decentralized control preserves availability at the expense of consistency [47]. LF enables quantified tradeoffs, where a bounded amount of inconsistency may be allowed in order to improve availability. Moreover, when networks misbehave sufficiently such that specified bounds on inconsistency or unavailability can no longer be respected, LF provides for the specification of application-specific fault handlers.

Next, we explain intuitively how both the centralized and decentralized coordination techniques work based on the federated aircraft door example in Figure 13. For the purposes of this demonstration, suppose that the two buttons for disarm and open are pressed simultaneously, such that the two physical actions observe the same physical time T when assigning a new tag g, where $\mathcal{T}(g) = T$, to the two scheduled events. In consequence, reactions 2 and 3 of `Cockpit` are logically simultaneous, but since reactions within the same reactor are mutually exclusive in LF, they execute in order, 2 before 3. Nonetheless, since these reactions are logically instantaneous, the outgoing `disarm` and `open` events will have the same tag g.

6.2.1 Centralized Coordination. In the centralized coordination technique, the RTI orchestrates the execution and takes control of when precisely a federate may process an event. Figure 14 depicts the full exchange of messages between the RTI and the (Cockpit), `Door`, and `Relay` federates for decentralized coordination.

Assuming that no events with tags earlier than g are present at the `Cockpit` federate, it will send a **Next Event Tag (NET)** message with tag g to the RTI to inform the RTI of its earliest tag at which it may send a message over the network. The RTI will in turn look at the structure of the federation with respect to the `Cockpit` federate and the latest reported state of federates to calculate the tag to which the `Cockpit` federate can advance to. In this case, this federate does not have any "upstream" federates that can send network messages. Therefore, it can advance its tag to g without later receiving a network message with an earlier tag. The advancement of tag is granted to the `Cockpit` federate by the RTI via a **Tag Advance Grant (TAG)** message that carries

the g tag in its payload. We note that due to the absence of any upstream federates, the Cockpit federate will in fact always be granted a TAG equal to its latest NET. Thus, as an optimization, the LF compiler will optimize this federate to advance its tag without waiting for a TAG from the RTI.

Subsequently, the Cockpit federate will advance its tag to g and will eventually send a disarm and an open message to the Relay and Door federates, respectively, and finish executing the reactions at the g tag. Next, it will send a **Logical Tag Complete (LTC)** message to the RTI with a g payload. Upon receiving the disarm and open messages, both the Relay and the Door federates will eventually send a NET message to the RTI carrying the g tag.

Deciding whether to send a TAG to the Relay federate is simple because it only has one upstream federate. Under our centralized coordination method, Relay will be granted to advance to g whenever its upstream federate (i.e., Cockpit) has sent a NET with a tag larger than g or an LTC with a tag equal to or larger than g to the RTI. Note the subtle requirement here that the RTI must receive the LTC from Cockpit before it can send a TAG to Relay.

In the case of the Door federate the decision whether to grant a tag advancement is more complicated. After receiving the open message with tag T, the Door federate will eventually send a NET message with the same tag to the RTI. However, the Door federate has two upstream federates, one of which might still be processing tag g (i.e., still deciding on whether the door should disarm). This is the critical point at which the RTI will prevent the execution of the open event before Relay gets the chance to complete its processing of disarm. Note that here, the RTI will grant a TAG for T to Door only if it has received an LTC with a tag g from Relay or has received a NET with a tag larger than g from Relay. Here lies another subtlety of our centralized coordination. Notice that for a deterministic execution under this coordination mechanism, the disarm message must arrive at Door before the RTI can grant a TAG to Door. In other words, if the TAG message arrives at Door from the RTI before the disarm message arrives from Relay, there is still a possibility of nondeterminism. To prevent this, all communication between federates, including the disarm message, go through the RTI. This requirement in combination with a strict ordering of network messages ensures that the RTI will be able to send the TAG message only after receiving the LTC message, which can only be received after the disarm message.

6.2.2 Decentralized Coordination. In the decentralized coordination technique, each federate operates on its and own decides when it is safe to process an event. This analysis relies on assumptions on the execution times of reactions as well as network latency and clock synchronization error.

Let the bound on the execution time of reactions 2 and 3 of Cockpit be X_2 and X_3, respectively. The disarm and open messages are launched into the network no later than Cockpit's physical times $T + X_2$ and $T + X_2 + X_3$, respectively. The dependence on X_2 in reaction 3 is a consequence of the fact that reaction 2 must execute before reaction 3 at any logical time.

Consider the lower message path. Suppose the network latency bound is L. The message arrives at the Door federate no later than time $T + X_2 + X_3 + L$, according to the physical clock at the Cockpit federate. Assume the bound on the clock synchronization error is E. Then the message arrives at Door no later than time $T + X_2 + X_3 + L + E$, according to the physical clock at the *Door* federate. On the logical timeline, this message still has tag g (where $\mathcal{T}(g) = T$). Upon receiving the message, the Door federate has to decide if the event is safe to process. But for this, we also have to examine the upper message path.

On the upper message path, there are two network hops and a reaction in the Relay federate that must be accounted for. Suppose that the Relay's reaction execution time is no larger than X_R. Then a similar analysis reveals that the upper message arrives at the Door federate no later than physical time $T + X_2 + 2L + X_R + E$ (note that Relay can process incoming messages immediately because it has only one input path). This message also has tag g (where $\mathcal{T}(g) = T$).

With this analysis, we can determine that any arriving message at either input port of the Door federate with tag g is *safe to process* when the physical clock at the Door federate reaches $S = \mathcal{T}(g) + \max\{X_2 + X_3 + L + E, X_2 + 2L + X_R + E\}$. All the Door federate needs to do, when receiving messages, is watch its local physical clock until that clock hits this precomputed threshold. If all the assumptions have been satisfied, the federate can be sure that it will not later see a message with an earlier tag. For the situation where both disarm and open were sent with the same tag g, Door can be sure that both messages have arrived by physical time S. Since the order of logical simultaneous reactions is deterministic, reaction 1 will execute before reaction 2, thus ensuring that the door is disarmed before it is opened.

The PTIDES technique implements the intuitively appealing solution that we suggested earlier: wait a while before opening the door. But PTIDES forces us to make explicit assumptions when we determine how long to wait. Since the connection topology of LF programs is known, the amount of time to wait can be precomputed based on that information. Moreover, messages do not need to flow through a centralized RTI, nor does it need to be consulted to advance time. As a consequence, there is no bottleneck and no single point of failure.

There are various techniques that can be used to improve on the above analysis. For example, the amount of time a federate has to wait can be reduced by designating a logical time delay on a connection between federates, as shown in Section 4.5. Any logical delay on the connection will simply be subtracted from the thresholds computed above. In addition, the dependence on the execution times of reactions in the path of a message can be reduced or eliminated by a deadline at the sender to ensure that messages are never launched into the network later than expected. Also, dynamically changing networks of reactors can be supported as long as the mutations occur at a well-defined logical time. The safe-to-process thresholds will need to be recomputed when such mutations occur. These optimizations, however, are beyond the scope of this article.

We can now see why the decentralized PTIDES technique emphasizes availability over consistency. Under this coordination mechanism, unlike centralized coordination, each federate advances logical time as its physical clock advances. As a consequence, it can continue to respond to local events even in the event of total network failure. If it later receives a message with a tag that is earlier than expected, given the assumptions, it invokes a fault handler because maintaining consistency is no longer possible.

Whether to emphasize consistency over availability or the other way around is application dependent. Consider, for example, an **Advanced Driver Assistant System (ADAS)** that augments the braking system of a car with vision-based automatic braking, possibly based in the cloud. If the brake pedal is pushed by the driver, it is imperative that the system reacts immediately regardless of the state of the network. Availability is more important than consistency. On the other hand, consider a four-way intersection with automated vehicles coordinating their use of it. Here, consistency is more important than availability.

6.3 Future Work

6.3.1 Verification. Nondeterministic behavior in software constructed using adjacent technologies like actors and publish-subscribe do not only make it more difficult to understand programs and test them rigorously; it hampers formal verification. Particularly for applications that are high in complexity and for which the cost of failure is high, such as cyber-physical systems, this poses a real concern. Moreover, formal verification normally requires a piece of software to be manually translated into a model that is amenable for verification, which, in turn, creates another verification problem (i.e., whether the constructed model is a faithful representation of the system). With LF, we are working toward the automatic generation of verification models and checking user-provided properties using **Bounded Model Checking (BMC)** through the verification engine

Uclid5 and the Z3 SMT solver. Early results indicate that LF programs reveal sufficient structure to construct axiomatic models from them that allow for the verification of useful safety properties.

6.3.2 Targeting Time-predicatable Hardware. For applications that are time critical, based on the promise of **PREdictable-Time (PRET)** hardware [20], LF provides a pathway toward a practice where real-time software can be written such that a compiler can provide it with static feasibility guarantees. In lieu of such hardware, deadline violations can still be detected at runtime in LF programs running on general-purpose CPUs. Yet, we believe that the time-centric programming model of LF lends itself especially well to the construction of programs for PRET machines and can possibly help make this kind of hardware more practical in its use.

6.3.3 Performance Analysis. Asynchronous programming models like actors have become known and appreciated because of their ability to exploit concurrency and scale well, particularly under I/O-intensive workloads. Our experimentation has shown evidence that LF's synchronous paradigm can compete with and even outperform reference implementations based on actors found in the Savina benchmark suite [35]. We leave it as future work to collect more evidence of this conjecture, but we have come to believe that the coordination approach we discuss in this article holds promise to guarantee determinism *and* scale well.

6.3.4 Runtime Mutations. At the moment, LF programs (including federated ones) require all reactor instances to be known and present at the start of the execution. We plan to develop mechanisms to make LF programs more dynamic. One avenue is to implement mutations [51], which are distinguished reactions that have the ability to modify the reactor's connection topology at runtime. Modal abstractions that have recently been developed for LF and its C and Python target could also help toward achieving this goal, albeit in a more limiting form.

7 RELATED WORK

Like Lingua Franca, a few other formalisms embrace a multiplicity of timelines in parallel and distributed systems. The MARTE profile of UML, and its Time Model and **Clock Constraint Specification Language (CCSL)** [56], specifies constraints among instants in a multiplicity of clocks. TimeSquare analyzes systems of constraints in CCSL [19]. CECAL can be used for embedded systems with distinct clocking mechanisms [65]. **Tagged Events Specification Language (TESL)**, like LF, uses explicit tags and ensures determinism [11]. Neither TESL nor CCSL is a programming language, but rather a language for modeling timing relationships. Deantoni et al. [18] have leveraged CCSL in GEMOC Studio [12] to create an omniscient debugger, trace validator, and assertion checker for LF.

Synchronous languages, especially Signal and Multiclock Esterel [9], explicitly support a multiplicity of abstract timelines. Signal supports asynchronous actions and nondeterministic merging of signals. Some care is required when comparing our work to these efforts, however. We use the term "clock" in a more classical way as something that measures the passage of physical time. In the synchronous language use of the term "clock," a sequence of events sent from one reactor to another has an associated "clock," which is the sequence of tags associated with those events. Since these clocks can all be different, LF supports at least the multiplicity of timelines, like those in Multiclock Esterel. A federated execution of LF also has the capability of decoupling logical time advancement across nodes, so despite our tags coming from a totally ordered set, LF achieves properties similar to the polychrony of Signal. LF can even accomplish the nondeterminism of Signal by using physical connections. Like LF, Signal can be used effectively to design distributed systems [24]. A major difference, however, is that LF is a coordination language, with the program

logic expressed in a target language (C, C++, or TypeScript), whereas Signal is a complete standalone programming language.

Like LF, Timed C [63] has a logical time that does not elapse during the execution of a function (except at explicit "timing points"). Moreover, like LF, priorities are inferred from timing information in the program. The deadlines of LF are all "soft deadlines" in the terminology of Timed C, meaning that the tasks are run to completion even if they will lead to a deadline violation. It would be useful further work to realize the "firm deadlines" of Timed C, but these require the use of low-level C primitives `setjmp` and `longjmp`, and it is not clear that it is possible to provide these for all the target languages that LF supports. A different and more generalizable approach to achieve preemption of reactions would be for the LF runtime to rely on an underlying thread scheduler with dynamic priorities. Real-time operating systems like FreeRTOS[5] and Zephyr[6] would be good candidate platforms to experiment with such priority-based preemptive scheduling.

8 CONCLUSION

We have shown that popular coordination approaches such as actors, publish-subscribe systems, and distributed shared memory are inadequate for delivering deterministic concurrency. The temporal model we discuss in this article exposes relationships between logical time and physical time across different observers in the system and serves as the basis of the deterministic concurrency model of reactors. In that model, logical time is decoupled from physical time except at points where the program interacts with its physical environment by explicitly invoking timers, physical actions, and deadlines. This approach 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. The goal of our reactor-oriented coordination language Lingua Franca is to strengthen mainstream programming languages with this capability and provide the tools for building robust, reliable, and testable concurrent systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Alain Girault (editor) for their helpful comments on earlier versions of this manuscript.

REFERENCES

[1] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. 2007. *Thrift: Scalable Cross-language Services Implementation*. Technical Report. Facebook.

[2] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. 1997. A foundation for actor computation. *Journal of Functional Programming* 7, 1 (1997), 1–72.

[3] S. Alimadadi, A. Mesbah, and K. Pattabiraman. 2016. Understanding asynchronous interactions in full-stack Javascript. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*. 1169–1180. https://doi.org/10.1145/2884781.2884864

[4] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in Erlang* (2nd ed.). Prentice Hall.

[5] AUTOSAR. 2019. Explanation of adaptive platform design. *AUTOSAR AP Release 19-11*.

[6] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang Meuter. 2013. A survey on reactive programming. *ACM Computing Surveys* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666

[7] Albert Benveniste and Gérard Berry. 1991. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE* 79, 9 (1991), 1270–1282.

[5] https://www.freertos.org/

[6] https://zephyrproject.org/

[8] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.5606

[9] G. Berry and E. Sentovich. 2001. Multiclock esterel. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*.

[10] Alessandro Biondi and Marco Di Natale. 2018. Achieving predictable multicore execution of automotive applications using the LET paradigm. In *IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'18)*. https://doi.org/10.1109/RTAS.2018.00032

[11] Frédéric Boulanger, Christophe Jacquet, Cécile Hardebolle, and Iuliana Prodan. 2014. TESL: A language for reconciling heterogeneous execution traces. In *ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE'14)*. https://doi.org/10.1109/MEMCOD.2014.6961849

[12] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution framework of the GEMOC studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE'16)*. Association for Computing Machinery, New York, NY, 84–89. https://doi.org/10.1145/2997364.2997384

[13] Christian Bradatsch, Florian Kluge, and Theo Ungerer. 2016. Data age diminution in the logical execution time model. In *International Conference on Architecture of Computing Systems (ARCS'16)*, Vol. LNCS 9637. Springer, 173–184. https://doi.org/10.1007/978-3-319-30695-7_13

[14] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *IEEE Computer* 45, 2 (February 2012), 23–29. https://doi.org/10.1109/MC.2012.37

[15] Adam Cataldo, Edward A. Lee, Xiaojun Liu, Eleftherios Matsikoudis, and Haiyang Zheng. 2006. A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems (WODES'06)*. http://ptolemy.eecs.berkeley.edu/publications/papers/06/constructive/

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's globally-distributed database. *ACM Transactions on Computer Systems (TOCS'13)*. 31, 8 (2013). DOI: https://doi.org/10.1145/2491245

[17] Fabio Cremona, Marten Lohstroh, David Broman, Edward A. Lee, Michael Masin, and Stavros Tripakis. 2017. Hybrid co-simulation: It's about time. *Software and Systems Modeling* (November 2017).

[18] Julien Deantoni, João Cambeiro, Soroush Bateni, Shaokai Lin, and Marten Lohstroh. 2021. Debugging and verification tools for LINGUA FRANCA in GEMOC studio. In *2021 Forum on Specification Design Languages (FDL'21)*. 1–8. https://doi.org/10.1109/FDL53530.2021.9568383

[19] Julien Deantoni, Frédéric Mallet, and Charles André. 2009. On the formal execution of UML and DSL models. In *WIP of the 4th International School on Model-driven Development for Distributed, Realtime, Embedded Systems*.

[20] Stephen A. Edwards and Edward A. Lee. 2007. The case for the precision timed (PRET) machine. In *Proceedings of the 44th Annual Conference on Design Automation (DAC'07)*. ACM, New York, NY, 264–265. https://doi.org/10.1145/1278480.1278545

[21] John C. Eidson. 2006. *Measurement, Control, and Communication Using IEEE 1588*. Springer.

[22] Clément Fournier. 2021. *A Rust Backend for Lingua Franca*. Diploma thesis. TU Dresden. https://cfaed.tu-dresden.de/publications?pubId=3247

[23] Richard Fujimoto. 2000. *Parallel and Distributed Simulation Systems*. John Wiley and Sons, Hoboken, NJ.

[24] Abdoulaye Gamatie and Thierry Gautier. 2010. The signal synchronous multiclock approach to the design of distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems* 21, 5 (2010), 641–657.

[25] David Gelernter. 1985. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 1 (1985), 80–112.

[26] Kai-Björn Gemlau, Leonie Köhler, Rolf Ernst, and Sophie Quinton. 2021. 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 (January 2021), 1–27. https://doi.org/10.1145/3381847

[27] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. 81–94.

[28] Arkadeb Ghosal, Thomas A. Henzinger, Christoph M. Kirsch, and Marco A. A. Sanvido. 2004. Event-driven programming with logical execution times. In *7th International Workshop on Hybrid Systems: Computation and Control (HSCC'04)*, Vol. LNCS 2993. Springer-Verlag, 357–371.

[29] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. 2001. Embedded control systems development with Giotto. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'01)*. Association for Computing Machinery, New York, NY, 64–72. https://doi.org/10.1145/384197.384208

[30] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. 2001. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, Vol. LNCS 2211. Springer-Verlag, 166–184.
[31] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. 2003. Giotto: A time-triggered language for embedded programming. *Proceedings of IEEE* 91, 1 (2003), 84–99. https://doi.org/10.1109/JPROC.2002.805825
[32] Carl Hewitt. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence* 8, 3 (1977), 323–363.
[33] Pierre-Emmanuel Hladik. 2018. A brute-force schedulability analysis for formal model under logical execution time assumption. In *ACM Symposium on Applied Computing (SAC'18)*. 609–615. https://doi.org/10.1145/3167132.3167199
[34] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. 2008. MQTT-S—A publish/subscribe protocol for wireless sensor networks. In *3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*. IEEE, 791–798.
[35] Shams M. Imam and Vivek Sarkar. 2014. Savina - An actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control (AGERE!'14)*. Association for Computing Machinery, New York, NY, 67–80. https://doi.org/10.1145/2687357.2687368
[36] IEEE Instrumentation and Measurement Society. 2002. *1588: IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. Report. IEEE.
[37] Florian Kluge, Martin Schoeberl, and Theo Ungerer. 2016. Support for the logical execution time model on a time-predictable multicore processor. *ACM SIGBED Review* 13, 4 (September 2016), 61–66. https://doi.org/10.1145/3015037.3015047
[38] Leonie Köhler, Phil Hertha, Matthias Beckert, Alex Bendrick, and Rolf Ernst. 2023. Robust cause-effect chains with bounded execution time and system-level logical execution time. *ACM Transactions on Embedded Computing Systems* 22, 3 (April 2023), 1–28. https://doi.org/10.1145/3573388
[39] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. 2005. *The Time-triggered Ethernet (TTE) Design*. IEEE.
[40] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. 1999. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall PTR.
[41] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563
[42] Leslie Lamport. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems* 6, 2 (1984), 254–280.
[43] Edward A. Lee. 2006. The problem with threads. *Computer* 39, 5 (2006), 33–42. https://doi.org/10.1109/MC.2006.180
[44] Edward A. Lee. 2008. Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'08)*. IEEE, 363–369. https://doi.org/10.1109/ISORC.2008.25
[45] Edward A. Lee. 2009. Computing needs time. *Communications of the ACM* 52, 5 (2009), 70–79. https://doi.org/10.1145/1506409.1506426
[46] Edward A. Lee. 2021. Determinism. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5 (July 2021), 1–34. https://doi.org/10.1145/3453652
[47] Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. 2021. Quantifying and generalizing the CAP theorem. *arXiv:2109.07771 [cs.DC]* (September 16 2021). https://arxiv.org/abs/2109.07771
[48] Edward A. Lee and Alberto Sangiovanni-Vincentelli. 1998. A framework for comparing models of computation. *IEEE Transactions on Computer-aided Design of Circuits and Systems* 17, 12 (1998), 1217–1229. http://ptolemy.eecs.berkeley.edu/publications/papers/98/framework/
[49] Edward A. Lee and Sanjit A. Seshia. 2017. *Introduction to Embedded Systems - A Cyber-physical Systems Approach* (2nd ed.). MIT Press, Cambridge, MA. http://LeeSeshia.org
[50] Marten Lohstroh. 2020. *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html
[51] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. 2019. 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, 27.
[52] Marten Lohstroh and Edward A. Lee. 2019. Deterministic actors. In *Forum on Specification and Design Languages (FDL'19)*.
[53] Marten Lohstroh and Edward A. Lee. 2020. A language for deterministic coordination across multiple timelines. In *Forum for Specification and Design Languages (FDL'20)*. IEEE, 1–8.

[54] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS), Special Issue on FDL'19* 20, 4 (May 2021), Article 36. https://doi.org/10.1145/3448128

[55] Oded Maler, Zohar Manna, and Amir Pnueli. 1992. From timed to hybrid systems. In *Real-time: Theory and Practice, REX Workshop*. Springer-Verlag, 447–484.

[56] Frédéric Mallet. 2008. Clock constraint specification language: Specifying clock constraints with UML/MARTE. *Innovations in Systems and Software Engineering* 4, 3 (2008), 309–314. https://doi.org/10.1007/s11334-008-0055-2

[57] Jorge Martinez, Ignacio Sañudo, and Marko Bertogna. 2018. Analytical characterization of end-to-end communication delays with logical execution time. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 37, 11 (November 2018), 2244–2254. https://doi.org/10.1109/TCAD.2018.2857398

[58] Eleftherios Matsikoudis and Edward A. Lee. 2013. An axiomatization of the theory of generalized ultrametric semilattices of linear signals. In *International Symposium on Fundamentals of Computation Theory (FCT'13)*, Vol. LNCS 8070. Springer, 248–258.

[59] Eleftherios Matsikoudis and Edward A. Lee. 2015. The fixed-point theory of strictly causal functions. *Theoretical Computer Science* 574 (2015), 39–77.

[60] Tobias R. Mayer, Lionel Brunie, David Coquil, and Harald Kosch. 2012. On reliability in publish/subscribe systems: A survey. *International Journal of Parallel, Emergent and Distributed Systems* 27, 5 (2012), 369–386.

[61] Christian Menard, Andrés Goens, Marten Lohstroh, and Jeronimo Castrillon. 2020. Achieving derterminism in adaptive AUTOSAR. In *Design, Automation and Test in Europe (DATE'20)*.

[62] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A distributed framework for emerging AI applications. *CoRR* abs/1712.05889 (2017). arXiv:1712.05889 http://arxiv.org/abs/1712.05889

[63] Saranya Natarajan and David Broman. 2018. Timed C: An extension to the C programming language for real-time systems. In *Real-time and Embedded Technology and Applications Symposium (RTAS'18)*. 227–239. https://doi.org/10.1109/RTAS.2018.00031

[64] Gerardo Pardo-Castellote. 2003. OMG data-distribution service: Architectural overview. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops, 2003.* IEEE, 200–206.

[65] Marie-Agnes Peraldi-Frati and Julien DeAntoni. 2011. Scheduling multi clock real time systems: From requirements to implementation. In *2011 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*. 50–57.

[66] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y. Ng, and others. 2009. ROS: An open-source robot operating system. In *International Conference on Robotics and Automation (ICRA'09)*. Workshop on Open Source Software.

[67] Stefan Resmerita, Andreas Naderlinger, and Stefan Lukesch. 2017. Efficient realization of logical execution times in legacy embedded software. In *ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'17)*. 36–45. https://doi.org/10.1145/3127041.3127054

[68] Raymond Roestenburg, Rob Bakker, and Rob Williams. 2016. *Akka in Action*. Manning Publications Co.

[69] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. 2013. 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)*. 75–82. https://doi.org/10.1109/VLHCC.2013.6645246

[70] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. 2018. Patmos: A time-predictable microprocessor. *Real-time Systems* 54(2) (April 2018), 389–423.

[71] Yang Zhao, Edward A. Lee, and Jie Liu. 2007. A programming model for time-synchronized distributed real-time systems. In *Real-time and Embedded Technology and Applications Symposium (RTAS'07)*. IEEE, 259–268. https://doi.org/10.1109/RTAS.2007.5

[72] Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. 2014. FlexPRET: A processor platform for mixed-criticality systems. In *2014 IEEE 19th Real-time and Embedded Technology and Applications Symposium (RTAS'14)*. IEEE, 101–110.

Received 15 January 2022; revised 21 April 2023; accepted 18 July 2023