A Language for Deterministic Coordination Across Multiple Timelines

Marten Lohstroh*, Christian Menard†, Alexander Schulz-Rosengarten‡,
Matthew Weber*, Jeronimo Castrillon†, and Edward A. Lee*
*EECS Department, UC Berkeley, USA
{marten, matt.weber, eal}@berkeley.edu
†Chair for Compiler Construction, TU Dresden, Germany
{christian.menard, jeronimo.castrillon}@tu-dresden.de
‡Department of Computer Science, Kiel University, Germany
als@informatik.uni-kiel.de

***Abstract*—We discuss a novel approach for constructing deterministic reactive systems that evolves around a temporal model which 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. What sets LF apart from other languages that treat time as a first-class citizen is that it confronts the issue that in any reactive system there are at least two distinct timelines involved; a *logical* one and a *physical* one—and possibly multiple of each kind. LF provides a mechanism for relating events across timelines, and guarantees deterministic program behavior under quantifiable assumptions.**

***Index Terms*—concurrency control, distributed computing, programming, software testing**

I. Introduction

Common software engineering approaches for expressing concurrent programs, such as threads [1], actors [2], [3], reactive programming [4], publish-subscribe systems [5], and even single-threaded event loops [6], make it difficult to achieve one important property of non-concurrent computing systems: *determinism*. Without a deterministic execution semantics, it quickly becomes intractable to rigorously test or verify the correctness of concurrent software. With the growing pervasiveness of networked computing and a trend towards integrating computationally demanding artificial intelligence components into real-time cyber-physical systems (think of robotics, autonomous vehicles, etc.) there is a need to achieve reliable and reproducible behavior in concurrent systems.

It has been shown that even for applications that do not pose 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 [7], [8], [9]. Google's Cloud Spanner [10], 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 [11], 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: Sec. II gives a motivating example showing how existing coordination paradigms fall short of delivering repeatable and testable concurrent behavior. Sec. III discusses the fundamental challenges and opportunities of using time for specifying concurrent behavior. Sec. IV introduces LF and explains how it allows a programmer to relate events across distinct timelines. Sec. V shows how the discrete event semantics of LF can be preserved across distributed components. Finally, Sec. VI discusses related work, and Sec. VII concludes.

II. 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.

This work was supported in part by NSF award #CNS-1836601 (Reconciling Safety with the Internet) and the Industrial Cyber-Physical Systems (iCyPhy) research center, supported by Denso, Siemens, and Toyota. This work was also supported in part by the Center for Advancing Electronics Dresden (cfaed), the German Academic Exchange Service (DAAD), and the KMU-innovativ project EM-RAM (01IS17016D) funded by German Federal Ministry of Education and Research (BMBF).

```
actor Door {
  closed = true;
  armed = true;
  handler disarm(){
    ... actuate ...
    armed = false;
  }
  handler open(arg){
    ... actuate ...
    closed = false;
  }
}
```

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

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

```
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 Fig. 1 yielding a nondeterministic program. The actor `Door` remains the same.

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 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], [3], as realized in Erlang [12], Akka [13], and Ray [14]. The pseudo-code example given in Fig. 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 Fig. 1, subject to the constraint that handlers are mutually exclusive, is deterministic [15]. However, this property breaks with even the slightest change to the actor network. Consider the minor elaboration shown in Fig. 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 Fig. 2 to consistently behave correctly [15].

In robotic systems, such as in ROS [16], or in the Internet of Things, such as in MQTT [17], publish-and-subscribe protocols are widely used to coordinate software components. Since such communication fabrics provide no assurances about the order of message delivery nor the order of message handling, they are prone to the same problem of nondeterministic behavior. ROS 2 uses DDS (Data Distribution Service) [18], 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.

Another approach could rely on a distributed shared memory architecture, often realized using the tuple space concept of Linda [19]. 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 [20]), are increasingly applied in cyber-physical systems (e.g., AUTOSAR Adaptive Platform [21]). But they, too, admit nondeterminism. Recent work shows that this nondeterminism can have fatal consequences in safety-critical applications [22] and presents a solution based on the same underlying principles that this paper builds upon.

The approach we advocate in this paper 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?

III. Reasoning About Time

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`

[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.

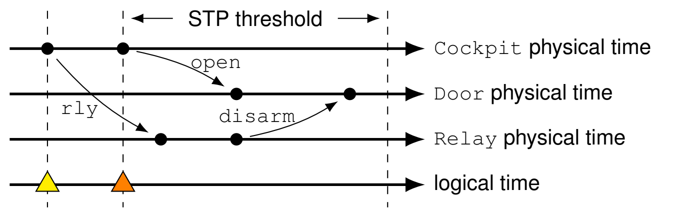

Fig. 3. Different observers may see events in a different order. An additional logical timeline allows to establish a global ordering. After a certain safe-to-process (STP) threshold, `Door` received all relevant messages and can use the logical timeline to determine that `disarm` should be processed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

messages originate from the same source, they may arrive out of order. The originator sees a different order from the recipient, as shown in Fig. 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 [23]. If messages carry timestamps, then our requirement can be that every actor processes messages in timestamp 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 [24]. 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 with 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. Fig. 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 [25], L [26], and X [27], [28].

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, quantified, and their violation detectable.

IV. Lingua Franca

The focus of LF is on network-integrated reactive and cyber-physical systems [29]. 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 [30]. 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 timestamp called a *tag* [31], and can carry a value of some datatype. Like signals in Esterel [32], events are present or absent at a given tag. Reactions are logically instantaneous, meaning that logical time does not advance during their execution. Setting the value of a port yields an event with a tag equal to the current logical time. Scheduling an action, on the other hand, yields an event with a tag strictly greater than the current logical time. Consequently, ports are a mechanism to 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.

A. Hello World

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

```
target C;
main reactor HelloWorld {
    reaction(startup) {=
        printf("Hello World!\n");
    =}
}
```

Each LF program starts with a target declaration that specifies the target language and may optionally contain further configuration. Currently, LF supports C, C++, and TypeScript 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 Sec. V), 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 $\{= \ldots =\}$ 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 nor parse the target code and instead relies on the target language compiler to perform language-specific checks.

B. 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.

1) Timers: Timers in LF 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 msec, 100 msec);
4     output y:int;
5     reaction(t) -> y {=
6         set(y, 42);
7     =}
8 }
```

The timer is named `t`, and its first triggering is 50 msec after the (logical) start of execution of the program. Subsequent triggers occur every 100 msec. 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 numerical relationships between logical time values (tags).

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 msec, 150 msec, 250 msec, etc. after the logical start time of the program.

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 `schedule` function, which takes two arguments, an action and a time offset. Consider the following reactor:

```
1  target C;
2  main reactor SlowingClock {
3      logical action a(100 msec);
4      state interval:time(100 msec);
5      reaction(startup) -> a {=
6          schedule(a, 0);
7      =}
8      reaction(a) -> a {=
9          printf("Logical time since start: \%lld
               nsec.\n", get_elapsed_logical_time());
10         schedule(a, self->interval);
11         self->interval += MSEC(100);
12     =}
13 }
```

This reactor produces its first output 100 msec after startup and then produces outputs with intervals that increase by 100 msec each time. The resulting output is this:

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

To accomplish this, the reactor defines on line 3 an action named `a` with a *minimum* delay of 100 msec. At startup, on line 6, the first reaction calls `schedule`, passing it the action `a`, and an *additional* delay of zero.

The second reaction (lines 8 to 12) is triggered by the action `a` and prints the elapsed logical time since execution start. It then calls `schedule` again, this time using the state variable named `interval` to specify an additional delay. It then increments the interval by 100 msec.

3) Logical vs. Physical Actions: The action `a` in the previous example is a *logical* action, which means that when a reaction calls `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 `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.

[2]The reason for a minimal delay specified separately from the extra delay passed to the `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 paper). The minimum delay is visible without parsing and analyzing target code.

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. 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 `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. Once such external events are assigned a tag, the order of further processing is determined exclusively by these tags.

C. 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          set(y, control);
8      =}
9  }
10 reactor Actuator {
11     input x:int;
12     reaction(x) {=
13         // Time-sensitive code
14     =} deadline(100 msec) {=
15         printf("*** Deadline miss detected.\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 `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 messages 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 msec on line 14 followed by a deadline violation handler. If this reaction is not invoked within 100 msec 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 that the calculation on line 6 (plus any overhead) not take more than 100 msec (in physical time). This relation across timelines is illustrated in Fig. 4.

The presence of such deadlines in the LF code enables the code generator to synthesize earliest-deadline-first scheduling policies. The fact that dependencies between reactions are also known to the code generator enables inheritance of the resulting priorities by all upstream reactions that may directly or indirectly trigger the reaction with a deadline.

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.

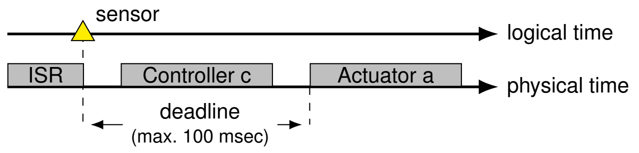

Fig. 4. A deadline defines the maximum delay between the logical time of an event and the physical time of the start of a reaction that it triggers.

D. 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 time line. 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 (LET) [33] to the producer, allowing the execution of the consumer to be timed more precisely with respect to physical time.

V. 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 (a

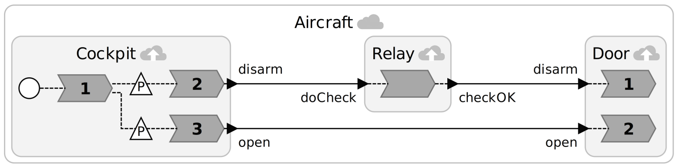

Fig. 5. A federated LF program implementing the aircraft door example. Each reactor may run on a different host machine.

"federate") can be mapped to a distinct host, giving rise to a distributed system. The communication within a federation can either be *centralized*, meaning that all exchanged messages flow through a central coordinator, or it can be *distributed*, meaning that all messages are exchanged directly between federates. In a distributed execution, the only role of the coordinator is to coordinate the start (and end) of execution, and possibly the process of new reactors joining the federations. The latter will require the use of runtime mutations (described in [30]), which are distinguished reactions that have the ability to modify the reactor's connection topology at runtime.

A. Federated Aircraft Door

A federated reactor that implements the aircraft door example is depicted in Fig. 5. The image is rendered automatically from LF code using the KIELER[3] Lightweight Diagrams framework [34]. In the diagram, reactors are denoted by boxes with rounded corners and reactions are denoted by chevron shapes. 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, it forwards the `disarm` message to `Door`.

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.

B. Coordinating Federated Execution

In the LF compiler, regular reactors are turned into federates by substituting their ports with reactors that are capable of sending and receiving messages over the network. By default, LF ensures that tags are preserved across the network using techniques that we will describe in the following. For some applications, however, there is no need to preserve tags on networked messages. For such applications, a connection between reactors can be designated to be a "physical connection," using the ~> operator instead of −>. 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 [24]. However, LF is not a simulation but an implementation language, which introduces unique problems.

One approach that we support uses a centralized controller called an RTI (Run Time Infrastructure). This is similar to several tools that implement the HLA standard (High Level Architecture) [35]. 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. This centralized approach, 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.

Another approach we support is the decentralized technique called PTIDES [8], which has none of these disadvantages. PTIDES, however, requires that the physical clocks on all federates be synchronized with some bounded error, using for example NTP or IEEE 1588 [36]. PTIDES also requires being able to bound network latencies and (certain) execution times. These three bounds (clock synchronization error, network latencies, and certain 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 [10].

C. PTIDES and the Aircraft Door

We can now explain intuitively how PTIDES works using the federated aircraft door example in Fig. 5. 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 $t = T$ to the two scheduled events. In consequence, reactions 2 and 3 are logical simultaneous, but reactions in LF are mutually exclusive and they execute in order, 2 before 3. Let the bound on execution time of these two reactions be X_2 and X_3, respectively. Since reactions are logically instantaneous, the outgoing `disarm` and `open` events have the same tag $t = T$. The two messages are launched into the network no later than physical times $T + X_2$ and

[3]https://rtsys.informatik.uni-kiel.de/kieler

$T+X_2+X_3$, 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 $t=T$. 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 $t=T$.

With this analysis, we can determine that any arriving message at either input port of the Door federate with tag t is *safe to process* when the physical clock at the Door federate reaches $S=t+\max\{X_2+X_3+L+E, X_2+2L+X_R+E\}$. 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 t, 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 Sec. IV-D. 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 paper.

VI. Related Work

Like LF, a few other formalisms embrace a multiplicity of time lines in parallel and distributed systems. The MARTE profile of UML, and its Time Model and CCSL (Clock Constraint Specification Language) [37] specify constraints among instants in a multiplicity of clocks. TimeSquare analyzes systems of constraints in CCSL [38]. CCSL can be used for embedded systems with distinct clocking mechanisms [39]. TESL (Tagged Events Specification Language), like LF, uses explicit tags and ensures determinism [40]. Neither TESL nor CCSL is a programming language, but rather a language for modeling timing relationships. They could prove useful for analyzing LF programs.

Synchronous languages, especially Signal and Multiclock Esterel [41], 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 advance, 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 [42]. 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 [43] 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 in our polyglot approach.

VII. Conclusion

We have shown that popular coordination approaches such as actors, publish-subscribe systems, and distributed shared memory are inadequate for delivering deterministic concurrency. While these asynchronous, nondeterministic coordination models may be favored for their ability to scale well, the coordination model we discuss in this paper holds promise to guarantee determinism *and* scale well. We have shown that understanding the relationships between logical time and physical time across different observers in the system is key to achieving this. As such, we argue that this relationship ought to be accessible in the programming model as a means to specify program behavior—not merely as an emergent property of its realization. The goal of our coordination language LF is to strengthen mainstream programming languages with this capability and provide the tools for building robust, reliable, and testable concurrent systems.

References

[1] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[2] C. Hewitt, "Viewing control structures as patterns of passing messages," *Journal of Artificial Intelligence*, vol. 8, no. 3, pp. 323–363, 1977.

[3] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, no. 1, pp. 1–72, 1997.

[4] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 52:1–52:34, Aug. 2013.

[5] T. R. Mayer, L. Brunie, D. Coquil, and H. Kosch, "On reliability in publish/subscribe systems: a survey," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 27, no. 5, pp. 369–386, 2012.

[6] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Understanding asynchronous interactions in full-stack javascript," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 1169–1180.

[7] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 2, pp. 254–280, 1984.

[8] Y. Zhao, E. A. Lee, and J. Liu, "A programming model for time-synchronized distributed real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2007, Conference Proceedings, pp. 259 – 268.

[9] E. A. Lee, "Computing needs time," *Communications of the ACM*, vol. 52, no. 5, pp. 70–79, 2009.

[10] J. C. Corbett *et al.*, "Spanner: Google's globally-distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 8, 2013.

[11] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[12] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent programming in Erlang*, 2nd ed. Prentice Hall, 1996.

[13] R. Roestenburg, R. Bakker, and R. Williams, *Akka In Action*. Manning Publications Co., 2016.

[14] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," *CoRR*, vol. abs/1712.05889, 2017.

[15] M. Lohstroh and E. A. Lee, "Deterministic actors," in *2019 Forum for Specification and Design Languages (FDL)*, Sep 2-4 2019, Conference Proceedings, pp. 1–8.

[16] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," vol. 3, 01 2009.

[17] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "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, 2008, pp. 791–798.

[18] G. Pardo-Castellote, "OMG data-distribution service: Architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE, 2003, pp. 200–206.

[19] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 80–112, 1985.

[20] A. Agarwal, M. Slee, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," Facebook, Tech. Rep., 4 2007.

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

[22] C. Menard, A. Goens, M. Lohstroh, and J. Castrillon, "Achieving derterminism in adaptive AUTOSAR," in *Design, Automation and Test in Europe (DATE 20)*, Grenoble, France, March 2020, in press.

[23] L. Lamport, R. Shostak, and M. Pease, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[24] R. Fujimoto, *Parallel and Distributed Simulation Systems*. Hoboken, NJ, USA: John Wiley and Sons, 2000.

[25] I. Instrumentation and M. Society, "1588: IEEE standard for a precision clock synchronization protocol for networked measurement and control systems," IEEE, Report, November 8 2002.

[26] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, "The time-triggered ethernet (TTE) design," in *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. IEEE, 2005, pp. 22–33.

[27] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: A processor platform for mixed-criticality systems," in *Real-Time and Embedded Technology and Application Symposium (RTAS)*, 2014.

[28] M. Schoeberl, W. Puffitsch, S. Hepp, B. Huber, and D. Prokesch, "Patmos: A time-predictable microprocessor," *Real-Time Systems*, vol. 54(2), pp. 389–423, Apr 2018.

[29] E. A. Lee, "Cyber physical systems: Design challenges," in *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2008, Conference Proceedings, pp. 363 – 369.

[30] M. Lohstroh, Í. Íncer Romeo, A. Goens, P. Derler, J. Castrillon, E. A. Lee, and A. Sangiovanni-Vincentelli, "Reactors: A deterministic model for composable reactive systems," in *8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy'19)*, vol. LNCS 11971. Springer-Verlag, 2019, Conference Proceedings, in press.

[31] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, 1998.

[32] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[33] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Embedded control systems development with Giotto," in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 64–72.

[34] C. Schneider, M. Spönemann, and R. von Hanxleden, "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)*, San Jose, CA, USA, Sep. 2013, pp. 75–82.

[35] F. Kuhl, R. Weatherly, and J. Dahmann, *Creating Computer Simulation Systems: an Introduction to the High Level Architecture*. Prentice Hall PTR, 1999.

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

[37] F. Mallet, "Clock constraint specification language: specifying clock constraints with UML/MARTE," *Innovations in Systems and Software Engineering*, vol. 4, no. 3, pp. 309–314, 2008.

[38] J. Deantoni, F. Mallet, and C. André, "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*, April 2009, Conference Proceedings.

[39] M. Peraldi-Frati and J. DeAntoni, "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*, 2011, pp. 50–57.

[40] F. Boulanger, C. Jacquet, C. Hardebolle, and I. Prodan, "TESL: A language for reconciling heterogeneous execution traces," in *ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*, October 2014, Conference Proceedings.

[41] G. Berry and E. Sentovich, "Multiclock Esterel," in *Correct Hardware Design and Verification Methods (CHARME)*, vol. LNCS 2144. Springer-Verlag, 2001, Conference Proceedings.

[42] A. Gamatie and T. Gautier, "The Signal synchronous multiclock approach to the design of distributed embedded systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 5, pp. 641–657, 2010.

[43] S. Natarajan and D. Broman, "Timed C: An extension to the C programming language for real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 11-13 2018, Conference Proceedings, pp. 227–239.