to appear in "Rebeca for Actor Analysis in Action," LNCS 15560, Springer, 2025

Verify Engineering Models, not Scientific Models

Shaokai Lin[0000−0001−6885−5572] and Edward A. Lee[0000−0002−5663−0584]

UC Berkeley, Berkeley, CA, 94720, USA
shaokai@berkeley.edu, eal@berkeley.edu

Abstract. Driving progress in science and engineering for centuries, models are powerful tools for understanding systems and building abstractions. However, the goal of models in science is different from that in engineering, and we observe the misuse of models undermining research goals. Specifically in the field of formal methods, we advocate that verification should be performed on engineering models rather than scientific models, to the extent possible. We observe that models under verification are, very often, scientific models rather than engineering models, and we show why verifying scientific models is ineffective in engineering efforts. To guarantee safety in an engineered system, it is the engineering model one should verify. This model can be used to derive a correct-by-construction implementation. To demonstrate our proposed principle, we review lessons learned from verifying programs in a language called Lingua Franca using Timed Rebeca.

Keywords: Formal Verfication · Lingua Franca · Rebeca.

Prologue

This paper is dedicated to our friend, colleague, and mentor, Marjan Sirjani, and offered as a contribution to her Festschrift. Marjan's work on formal methods for software engineering has been and continues to be an inspiration.

1 Scientific and Engineering Models

In science, the objective of a model is to match a real-world system. In contrast, in engineering, the objective of a real-world system (an implementation) is to match a model [15]. A digital logic diagram, for example, is an engineering model; a circuit gets constructed to match the model. Differential equations giving currents and voltages in a circuit over time are commonly used as a scientific model; the equations are crafted to match the physical circuit. But even these equations could be used as an engineering model, giving for example a specification of an analog circuit. The actual components used will not be perfectly linear and will not have the resistances, capacitance, and inductance perfectly matching the model. When using models, it is important to understand whether they are being used as engineering models or scientific models. Are the resistances wrong in the model or in the physical circuit? One of the two must be true.

This paper argues that formal verification, by its very nature, should focus on verifying engineering models, not scientific ones. As George Box famously said, "all models are wrong, but some are useful" [6]. Here, Box was referring to scientific models. The corresponding statement for engineering models is, "all physical, real-world implementations are wrong, but some are useful" [14]. An engineering model is, or at least attempts to be, right, not wrong. It is a specification, an idealization, and an implementation can only approximate it.

Formal verification, by its nature of being formal, can only make assertions about *models*. For scientific models, therefore, since the model itself is always wrong, proving "correctness" of a wrong model has questionable value. On the other hand, proving that a design (a specification) is correct can be extremely useful. Indeed, most success stories for formal verification concern synchronous digital logic (an engineering model) and software (another engineering model). The physical realization in both cases is "electrons sloshing in silicon" [15], about which nothing is proved.

In this paper, we examine a major innovation of Sirjani and her colleagues, Timed Rebeca (TR), a language and formalism for describing and analyzing timed systems. We argue that the timing features of TR are more useful when viewed as an engineering model than when viewed as a scientific model. To support the case, we show the relationship between the timing constructs of TR and an implementation (vs. modeling) language with similar timing constructs, Lingua Franca (LF). We show that, with the addition of priorities for message handlers, TR can model the timing constructs of Lingua Franca and therefore can be used to effectively formally verify engineering models specified using LF.

For example, the TR `after` delay gives a time offset between the sending and the receiving of a message between actors. If this `after` delay is intended to model physical communication latencies, then it is being used as a scientific model. The model is correct only if the physical communication latencies actually match. If, on the other hand, it is used to specify a logical offset between the sender's view and the receiver's view of changing data, then it is being used as an engineering model. The implementation is correct only if it actually delivers the specified logical offset. Such delays are used in LF in the latter sense, not the former, whereas in TR, we have seen them used both ways.

A significant limitation remains, however. Formal verification can only be performed on closed models. For systems that interact with an uncontrolled environment that is not designed, we have no choice but to provide a scientific model of that environment. Any proof of the correctness of a design, therefore, is predicated on the assumptions about the environment. These assumptions are made evident by making a clear separation between models of the environment and models of the system under design. The weaker those assumptions, the stronger the confidence provided by the verification results.

2 Scientific and Engineering Models in Timed Rebeca

Timed Rebeca [21] is an extension of the Rebeca modeling language [22] (an implementation of the Hewitt-Agha actor model [8, 1, 2]) that adds a notion of time. In this extension, time is an integer, messages are sent and arrive at (possibly distinct) times and message handlers (optionally) take time to execute. Three operators are added to the Rebeca language to model time: `delay`, `deadline`, and `after`. A delay statement models the passing of time of an actor during the execution of a message server. The `after` operator indicates time that elapses before a message is delivered to its destination. The `deadline` operator models message loss if the destination is busy due to `delay` statements and does not handle an incoming message before the deadline expires.

In addition to these operators, Timed Rebeca strengthens the semantics that define the order in which message handlers are invoked. Because of these operators, each message delivered to an actor has a time associated with it, and the destination actor will handle messages in temporal order, unless it is busy due to a `delay` statement, in which case it will handle the message at the time the delay expires. If two messages are delivered with the same time, then the handler order is nondeterministic unless the programmer specifies priorities. The ability to specify priorities was added later specifically to support the kind of modeling we describe in this paper [24, 9].

Reynisson et al. [21] give a ticket service example (Fig. 1), where an agent accepts requests for tickets from an environment and issues requests for a ticket to two ticket services. If the first ticket service does not respond with a ticket within a specified amount of time (defined by the `checkIssuedPeriod` environment variable, defined on line 1 and used on line 14), then it tries the second ticket service. The `TicketService` actor uses the `delay` operator to nondeterministically delay responding by either `serviceTime1` or `serviceTime2` (lines 38 and 39). Here, there is a mix of scientific and engineering modeling. The value of `checkIssuedPeriod` is an engineering *specification*. It defines how long the `Agent` actor should wait before giving up on the requested ticket service. On the other hand, the `serviceTime1` or `serviceTime2` variables model communication and computation time, which are *assumptions* derived from the environment through scientific means (measurement, for example). When performing verification on this model, the results of the verification can only be considered valid for these two particular assumed delays. If the actual observed delays deviate from these assumed delays, which could always happen due to the nature of scientific modeling, the verification model is no longer applicable.

Later, Sirjani and Khamespanah note the distinction between engineering models and scientific models and describe a Timed Rebeca example for components of a network on chip [23]. Specifically, the model consists of a Manager, which generates network traffic, and a Router, which routes traffic according to a specified policy. In this example, the precise times given in `after`, `delay`, and `deadline` statements can be interpreted as specifications for the time taken by a synchronous digital circuit implementation of the router. No time units are given in the paper, but in this engineering-model interpretation, they could be

```
env int requestDeadline,
     checkIssuedPeriod,
     retryRequestPeriod,
     newRequestPeriod, serviceTime1,
     serviceTime2;

reactiveclass Agent {
  knownrebecs { TicketService ts1;
       TicketService ts2; }
  statevars { int attemptCount;
    boolean ticketIssued; int token; }
  msgsrv initial() {
    self.findTicket(ts1); }
  msgsrv findTicket(TicketService ts){
    attemptCount += 1; token += 1;
    ts.requestTicket(token) deadline(
         requestDeadline);
    // Check if a reply to the
    // request has been received.
    self.checkTicket() after(
         checkIssuedPeriod);}
  msgsrv checkTicket() {
    if (!ticketIssued
      && attemptCount == 1) {
      self.findTicket(ts2);
    } else if (!ticketIssued
      && attemptCount == 2) {
      self.retry() after(
           retryRequestPeriod);
    } else if (ticketIssued) {
      ticketIssued = false;
      self.retry() after(
           newRequestPeriod);
    }
  }
  msgsrv ticketIssued(int tok){
    if (token == tok) {
      ticketIssued = true; }}
  msgsrv retry() {
    attemptCount = 0;
    self.findTicket(ts1);}
}
reactiveclass TicketService {
  knownrebecs { Agent a; }
  msgsrv initial() { }
  msgsrv requestTicket(int token){
    // The ticket service sends
    // the reply after a non-
    // determinstic delay of
    // either serviceTime1 or
    // serviceTime2.
    int wait = ?(serviceTime1,
         serviceTime2);
    delay(wait);
    a.ticketIssued(token);
  }
}
main {
  Agent a(ts1, ts2):();
  TicketService ts1(a):();
  TicketService ts2(a):();
}
```

Fig. 1. Timed Rebeca model for a ticket service, from [21].

clock cycles or SI units (such as nanoseconds) as measured by the clock of a synchronous digital circuit.

We argue here that the Timed Rebeca constructs, `delay`, `deadline`, and `after`, are much more useful in an engineering model than in a scientific model, except when explicitly modeling an uncontrolled environment. Using `after` to model communication delays, for example, is questionable for systems using modern network communication. The delays can be nondeterministically chosen, but for verification to be tractable, a small number of possible delays must be used. Moreover, these delays refer to clock measurements of time at distinct points in the network. Reynisson et al. [21] assume perfect (or near perfect) clock synchronization, but perfect clock synchronization is not possible. On the other hand, when these numbers refer to a *logical* clock instead of physical clocks, and the time ordering of events is a *semantic* property of the system rather than an emergent property of the implementation, then verification results can be trusted subject only to the condition that the system implementation correctly realizes the semantics. We show how to do that in this paper.

Fig. 2. Two accidental deployments of emergency escape slides (image from *The Telegraph*, Sept. 9, 2015).

3 Running Example

In this section, we introduce an example (somewhat artificial and grossly oversimplified) which is a distributed system in which the order of events is of critical importance. There are many practical examples that have this character, but we pick this one because it is vivid, simple, and challenging to get right, all at the same time.

Consider automating the opening and closing of an aircraft door. Modern passenger planes are equipped with emergency escape slides that are automatically deployed if the door is opened while the door is "armed." When the doors are closed for departure, flight attendants arm them. At arrival, once they have verified that a ramp is right outside the door, the flight attendants disarm the door before an operator opens it. The process currently is manual, with a red or green flag displayed in a window on the door to indicate whether it is armed or not.

Suppose we wish to automate the process so that the door can be opened remotely. There might be multiple systems on the aircraft that issue a command to open the door. Some of these, such as a cockpit control, might want to ensure that a ramp is present and the door is disarmed before it is opened, while others, such as a fire alarm system, might want to open the door whether it is armed or not.

The door will be equipped with a networked microprocessor that provides just two services, `disarm` and `open` (we leave out arming and closing, which are not needed to make our points). Assume the door starts in the armed and closed state. What should the microprocessor do when it receives an `open` command over the network? It is critical to be sure that if the `open` command has an associated `disarm` command, or if there is a ramp present at the door, that the door should be disarmed before it is opened. We have to guard against errors

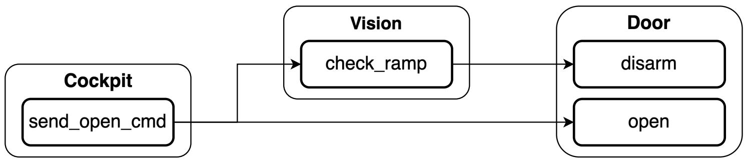


Fig. 3. Aircraft door system schematic using the Actor model.

```
 1 reactiveclass Cockpit(10) {
 2   knownrebecs { Vision v; Door d; }
 3   Cockpit() {
 4     self.send_open_cmd();
 5   }
 6   msgsrv send_open_cmd() {
 7     // Send open message.
 8     d.open(true);
 9     v.check_ramp(true);
10     // Repeat after 100 ms.
11     self.send_open_cmd() after(100);
12   }
13 }
14 reactiveclass Vision(10) {
15   knownrebecs { Door d; }
16   msgsrv check_ramp(boolean trigger) {
17     // Use vision to detect ramps.
18     // Assume a ramp is present.
19     d.disarm(true);
20   }
21 }
22 reactiveclass Door(10) {
23   statevars {
24     // 0: closed_armed,
25     // 1: closed_disarmed,
26     // 2: open,
27     // 3: open_deployed
28     int status;
29   }
30   Door() { status = 0; }
31   @priority(1)
32   msgsrv disarm(boolean disarm) {
33     // Disarm the door.
34     if (status == 0) status = 1;
35   }
36   @priority(2)
37   msgsrv open(boolean open) {
38     // Actuate door opener.
39     if (status == 0)
40       status = 3;
41     else if (status == 1)
42       status = 2;
43   }
44 }
45 main {
46   @priority(1) Cockpit c(v, d):();
47   @priority(2) Vision  v(d)   :();
48   @priority(3) Door    d()    :();
49 }
```

Fig. 4. Timed Rebeca model for the aircraft door system using priorities.

due to networking or computation delays, otherwise we might find ourselves in the life-threatening situations depicted in Fig. 2.

Suppose that when a cockpit control issues an instruction to open the door, the actual opening is delayed until a ramp is present. As shown in Fig. 3, a computer vision system might check for a ramp. Suppose further that the Cockpit, Vision, and Door components are realized as Hewitt-Agha actors. We immediately have a problem because the order of message delivery and handling in the Hewitt-Agha model is nondeterministic. It is highly sensitive to delays in the network and in intermediate computations. With the design in Fig. 3, the `open` message will very likely arrive before `disarm` message, resulting in catastrophe.

Fig. 4 shows a Timed Rebeca model that uses a combination of temporal semantics and explicit priorities to ensure correct ordering of message handlers. The constructor for the Cockpit component starts things off by sending itself a `send_open_cmd` on line 4. It then repeatedly invokes this same message handler every 100 time units on line 11. This, effectively, models an environment, where the time units for the repeat may represent a worst case maximum frequency for events. We have, therefore, a scientific model of an environment that encodes an assumption about its worst case behavior. To trust any verification results with this model, we must convince ourself that any time intervals larger than this will not result in violations of the properties of interest.

In this (simplified) model, the Door actor has a state variable `status` (line 28) that takes on the value 3 if the door opens and the slides are deployed (Fig. 2). Since this model only includes the cockpit control, and it assumes that the Vision actor always sees a ramp (line 19), then we would like to verify that `status` never reaches value 3. Using Afra, the Timed Rebeca model-checking tool, we can verify that this requirement is satisfied. However, the requirements would not be satisfied without the `@priority` annotations on lines 31, 36, 46, 47, and 48. The priority annotations are a recent addition in Timed Rebeca [24, 9]. They can be specified at the level of message servers (lines 31, 36) and at the level of actor instances (lines 46, 47, and 48). Semantically, a smaller priority value indicates higher execution priority. Components with the same priority get executed in a nondeterministic order when they are enabled. Priorities specified on message servers are treated separately from priorities specified on actors.

In this model, the only temporal operator is on line 11. Hence, all message transport and message handling is assumed to be instantaneous. Without the priorities, there is a risk that the `open` message handler of Door will be invoked before the `check_ramp` handler of Vision or, if not, before the `disarm` handler of Door. This, again, would result in catastrophe.

In a distributed system, enforcing these priorities is not trivial. The Lingua Franca language, described in the next section, infers priorities based on data dependencies and enforces them at runtime. Without LF, a distributed implementation of this design will be quite difficult to realize. Moreover, the enforcement aspect of the implementation will be error prone, weakening the verification results. We may have formally verified the design, but without a strong assurance that the implementation matches the design, the verification result is not very useful.

Given the temporal semantics of Timed Rebeca, it is tempting to instead model the timing of an implementation and verify that model. It may be, for example, that instead of assuming unrealistic instantaneous computation and message transport, the delays experienced in a physical implementation will lead to a safe design. The Timed Rebeca model in Fig. 5 attempts to check this. To open the aircraft door, the `Cockpit` actor calls the `send_open_cmd` handler, which sends two messages. It first sends a `check_ramp` message to the Vision actor (line 11) using an `after` delay to model the network delay. It then waits 5 time units (line 13) before sending an `open` message (line 16), again using an `after` delay to model the network delay. The Vision actor models its computation time on line 25 and the network delay of its relay to Door on line 29. Again, we can use Afra to verify that the Door actor never reaches `status == 3`.

```
 1 reactiveclass Cockpit(10) {
 2   knownrebecs {
 3     Vision v; Door d;
 4   }
 5   Cockpit() {
 6     self.send_open_cmd();
 7   }
 8   msgsrv send_open_cmd() {
 9     // Send message to check ramp.
10     // Max network latency = 2ms.
11     v.check_ramp(true) after(2);
12     // Delay to avoid accidents.
13     delay(5);
14     // Send open message.
15     // Max network latency = 2ms.
16     d.open(true) after(2);
17     // Repeat after 100 ms.
18     self.send_open_cmd() after(100);
19   }
20 }
21 reactiveclass Vision(10) {
22   knownrebecs { Door d; }
23   msgsrv check_ramp(boolean trigger) {
24     // Computation time = 2 ms.
25     delay(2);
26     // Use vision to detect ramps.
27     // Assume a ramp is present.
28     // Max network latency = 2
29     d.disarm(true) after(2);
30   }
31 }
32 reactiveclass Door(10) {
33   statevars {
34     // 0: closed_armed,
35     // 1: closed_disarmed,
36     // 2: open,
37     // 3: open_deployed
38     int status;
39   }
40   Door() { status = 0; }
41   msgsrv disarm(boolean disarm) {
42     // Disarm the door.
43     if (status == 0) status = 1;
44   }
45   msgsrv open(boolean open) {
46     // Actuate door opener.
47     if (status == 0)
48       status = 3;
49     else if (status == 1)
50       status = 2;
51   }
52 }
53 main {
54   Cockpit c(v, d):();
55   Vision  v(d)   :();
56   Door    d()    :();
57 }
```

Fig. 5. Using `after` and `delay` to avoid accidentally deploying emergency slides, model network latencies, and model execution time.

This model, however, exposes a critical flaw in this style of design. If we assume the `after` delays are worst-case scientific models of network delay, then we have a problem. If the `open` message from the Cockpit happens to experience less delay, say one time unit instead of two, then the model becomes nondeterministic, and the door may be opened before it is disarmed. To correct for this, we could increase the delay on line 13. In fact, this model mixes scientific models (lines 11, 16, 25, and 29) with engineering modeling (line 13). The scientific models must be interpreted as worst-case behaviors, and we have to do careful reasoning, separate from the formal verification, to assure ourselves that lower values will not hurt. On the other hand, the engineering model on line 13 has the opposite semantics. It must be interpreted as a lower bound, a *requirement* on the system design to ensure that accidental deployment of emergency slides never occurs. The formal verification performed by Afra does not help with this reasoning but only tells us whether the desired condition is satisfied for the particular delays chosen.

The ultimate root cause of the problem is mixing *engineering* models with *scientific* models in an undisciplined way. We illustrate how to correct this by clearly separating the model of the environment from the model of the system under design, and by exclusively using engineering models for the system under design.

4 Overview of Lingua Franca

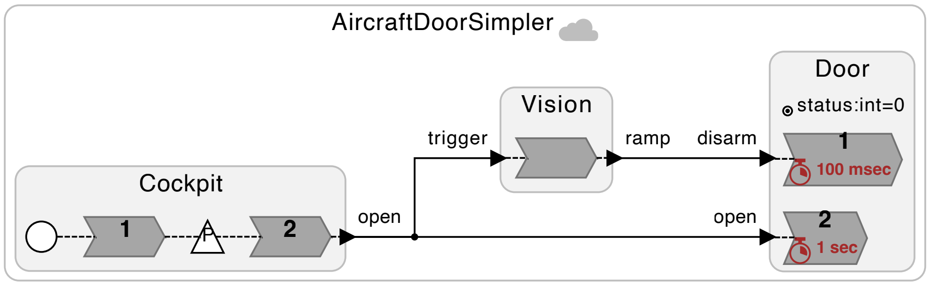
Lingua Franca (LF) [19] is a polyglot coordination language designed to augment multiple mainstream programming languages (also called target languages), currently C, C++, Python, TypeScript, and Rust, with deterministic reactive concurrency and the capability to specify timed behavior. LF is supported by a runtime system that enables concurrent and distributed execution of reactive programs, which can be deployed on various platforms, including in the cloud, at the edge, in containers, and even on resource-constrained bare-metal embedded platforms.

A Lingua Franca program defines interactions between components known as reactors [17], with the logic for each reactor written in plain target code. Lingua Franca's code generator then produces one or more programs in the target language, which are compiled using standard toolchains. When the application has parallelism, it runs on multiple cores without losing determinism. For distributed applications, multiple programs and scripts are generated to deploy these programs on multiple machines and/or containers. The network communication fabric connecting these components is also synthesized as part of the code generation and compilation process.

4.1 Reactor-Oriented Programming

Lingua Franca programs consist of reactors, which are stateful entities with event-driven routines. Reactors adopt advantageous semantic features from established models of computation, namely actors [3], logical execution time [10], synchronous reactive languages [5], and discrete event systems [13] (such as DEVS [25] and SystemC [16]). The reaction routines belonging to reactors can process inputs, generate outputs, alter the reactor's state, and schedule future events. Reactors resemble actors [3], which are software components that communicate through message passing. However, unlike traditional actors, these messages have timestamps, and the concurrent interaction of reactors is deterministic by default. Any nondeterministic behavior must be explicitly programmed if needed.

Fig. 6 shows a Lingua Franca program realizing the aircraft door service described in the previous section. The textual code shown at the bottom of the figure is Lingua Franca code, and the diagram above it is automatically generated by the tools and updated dynamically as the code is edited. In this example, the main program is *federated* (line 2), which means that each of the top-level reactors, `Cockpit`, `Vision`, and `Door`, will be code generated into its own program in the target language (specified on line 1). These programs, called *federates*, can be containerized for better isolation and fault tolerance and/or distributed to distinct hardware, for example to exploit specialized hardware in



```
 1 target C // or Python, Rust, etc.
 2 federated reactor {
 3   c = new Cockpit()
 4   v = new Vision()
 5   d = new Door()
 6   c.open -> d.open
 7   c.open -> v.trigger
 8   v.ramp -> d.disarm
 9 }
10 reactor Cockpit {
11   physical action command
12   output open: bool
13   reaction(startup) -> command {=
14     // Set up cockpit UI controls.
15     // Target language code here...
16   =}
17   reaction(command) -> open {=
18     // Target language code here.
19     // E.g., send open message.
20     lf_set(open, true);
21   =}
22 }
23 reactor Vision {
24   input trigger: bool
25   output ramp: bool
26   reaction(trigger) -> ramp {=
27     // Target language code here...
28   =}
29 }
30 reactor Door {
31   input open: bool
32   input disarm: bool
33   state status: int = 0
34   // This reaction should be first
35   reaction(disarm) {=
36     // Disarm the door.
37     if (self->status == 0) {
38       self->status = 1;
39     }
40   =} deadline(100 ms) {=
41     // Handle the deadline violation
42   =}
43   reaction(open) {=
44     // Actuate door opener.
45     if (self->status == 0) {
46       self->status = 3;
47     } else if (self->status == 1) {
48       self->status = 2;
49     }
50   =} deadline(1 s) {=
51     // Handle the deadline violation
52   =}
53 }
```

Fig. 6. Aircraft door system schematic and Lingua Franca specification.

the Vision reactor. This Vision reactor accesses a camera with a computer vision system that detects a ramp outside the door and disarms the door before opening it.

The grey chevrons in inside reactors in the diagram represent *reactions*, which are triggered by inputs, timers, or events on an event queue and are able to produce outputs. The business logic inside reactions is written in the target language that is chosen. A typical implementation of these reactors could use, for example, the Python target, which can leverage existing packages such as Tensorflow lite for the vision component, if it is to be realized on the edge, or it could use an API to realize the vision component in the cloud, or it could use the C target together with OpenCV, the open computer vision library (this code would go on line 27). Legacy code and libraries are easy to use in LF by simply invoking their APIs in the reaction bodies.

In principle, a federated LF program could use different target languages for each of the federates, such as Python for the `Vision` federate and C or Rust for the other components, but this capability only exists currently in concept demonstration form.

`Cockpit` uses a common pattern for LF programs that react to asynchronous events from the environment. It has a *physical action*, defined on line 11, which is used to inject asynchronous external events from the environment. The reaction to the *startup* event on line 13 can be used to set up any external interactions, for example enabling an interrupt service routine to handle sensor events or cockpit switches. When these events occur, the service routine schedules the physical action, which gets assigned a timestamp drawn from a local physical clock. The code on line 17 will then be invoked in reaction to such events.

A key property of Lingua Franca is that every reactor handles events in timestamp order, and when events bear the same timestamp, they will appear simultaneously at the reactor. Moreover, reactions are logically instantaneous. If they produce outputs in reaction to inputs, as the `Vision` does, those outputs have the same timestamp as the input. Consequently, LF semantics assures that the `Door` reactor will not react to an open command until the results of the `Vision` reactor are delivered to it. Moreover, when a reactor is presented with multiple simultaneous inputs, reactions will be invoked in the lexical order. Hence, given simultaneous `disarm` and `open` events, the `Door` reactor will react first to the `disarm` event (line 35) and then to the `open` event (line 43). As we have seen in the previous section, it is not trivial to ensure such behavior in an actor network, even one extended with time, as in Timed Rebeca. It is, nevertheless, possible, as we will show.

The `Door` reactor also includes two *deadline* declarations, which serve two purposes. First, they guide the LF scheduler to prioritize reaction invocations with nearer deadlines. Second, they provide fault handling code, on lines 40 and 50, which is invoked instead of the regular reaction code if and when the deadline is violated.

In this example, the federated reactors can be deployed on separate machines and reactions in different reactors can be executed by multiple threads in parallel. More importantly, for the same given input values and timing, the partial order constraints on the execution of reactions ensures determinism. This modular design of reactors, determinism, and flexibility in deployments make LF suitable for describing time-sensitive applications.

4.2 Specifying Timing Behavior

Lingua Franca makes a distinction between two timelines, **logical time** and **physical time** [18]. **Logical time** is represented by a tag (timestamp and microstep in superdense time [7, 20]) that tracks the processing of events within the system. It is a marker for the sequence and timing of events as understood by the system's logic. **Physical time** tracks the actual movement of physical clocks, not to be confused with the conceptual clocks used in synchronous-reactive models.

In LF's model of time, logical time, by default, lags behind physical time, meaning that the system's logical processing waits for the physical time to advance before proceeding.

Lingua Franca programs can explicitly specify a variety of timing behaviors in time-sensitive systems. These timing behaviors include time-triggered events, constraints on asynchronous events (such as minimum spacing), and deadlines. The timing behavior specification in Lingua Franca allows deterministic execution for timed events and user inputs. Reactions to events are executed in order based on the logical time (i.e., the tag).

The *physical action* in the `Cockpit` reactor (line 11), represented in the diagram by a triangle with a "P", captures external asynchronous inputs, assigning them a logical timestamp based on the physical time, as measured by a local clock. Lines 40 and 50 give deadlines, which specify a maximum acceptable gap between the logical time and physical time of a reaction invocation. These are also shown in the diagram as red markers in the reactions of the `Door` reactor.

Like Timed Rebeca, Lingua Franca supports **after delays** on connections between reactors (not used in this example). These increment the timestamp of an event conveyed along the connection. They are part of the program logic, not a scientific model of communication delays. This manipulation can be used to ameliorate that effects of the fundamental tradeoff between consistency and availability in a distributed system [12].

For federated programs, Lingua Franca offers two distinct coordination strategies [4]. The default coordinator is a centralized component called the RTI (for runtime infrastructure) that guarantees that events are processed in the proper semantic order regardless of communication delays, computation time, or clock synchronization. An alternative coordinator is decentralized and guarantees proper semantic order only under clearly stated assumed bounds on communication latency, computation times, and clock synchronization error. This coordinator also provides hooks to handle faults that occur when these assumptions are violated. Deadline statements in LF can also be viewed as clearly stated assumptions coupled with fault handlers to deal with violations of those assumptions.

In short, Lingua Franca offers deterministic behavior under clearly stated assumptions, mechanisms to detect when these assumptions are violated, and fault handlers so that applications can react appropriately to violation of the assumptions. This determinism applies even with parallel and distributed execution of LF programs, bringing the key advantages of determinism [11]: repeatability, consensus, predictability (sometimes), fault detection, simplicity, unsurprising behavior, and composability. For applications that require (or benefit from) nondeterminism, LF includes explicitly nondeterministic constructs that can be used. This is a notable contrast with most other concurrent and distributed computing frameworks, which give you nondeterminism by default and leave it to the designer to build deterministic behavior when needed.

5 Verifying an Engineering Model using Timed Rebeca

```
reactiveclass Cockpit(10) {
  knownrebecs {
    Door door;
    Vision Vision;
  }
  statevars { boolean open_value; }
  Cockpit() {
    // Schedule a physical action.
    self.reaction_1();
  }
  @globalPriority(2)
  msgsrv reaction_1() {
    open_value = true;
    door.read_port_open(open_value)
         after(0);
    Vision.read_port_trigger(
         open_value) after(0);
  }
}
reactiveclass Vision(10) {
  knownrebecs { Door door; }
  statevars {
    boolean ramp_value;
    boolean trigger_value;
    boolean trigger_is_present;
  }
  Vision() {
    trigger_is_present = false;}
  @globalPriority(4)
  msgsrv reaction_1() {
    // Assume a ramp is detected
    ramp_value = true;
    door.read_port_disarm(ramp_value
         ) after(0);
  }
  @globalPriority(3)
  msgsrv read_port_trigger(boolean
       _trigger_value) {
    trigger_value = _trigger_value;
    trigger_is_present = true;
    self.reaction_1();
  }
}
```

```
reactiveclass Door(10) {
  statevars {
    int status;
    boolean open_value;
    boolean open_is_present;
    boolean disarm_value;
    boolean disarm_is_present;
  }
  Door() {
    status = 0;
    open_is_present = false;
    disarm_is_present = false;
  }
  @globalPriority(6)
  msgsrv reaction_1() {
    if (status == 0) status = 1;
    disarm_is_present = false;}
  @globalPriority(8)
  msgsrv reaction_2() {
    // Deploy emergency slides.
    if (status == 0) status = 3;
    else if (status == 1) {
      status = 2;}
    open_is_present = false;}
  @globalPriority(7)
  msgsrv read_port_open
    (boolean _open_value) {
    open_value = _open_value;
    open_is_present = true;
    self.reaction_2();}
  @globalPriority(5)
  msgsrv read_port_disarm
    (boolean _disarm_value) {
    disarm_value = _disarm_value;
    disarm_is_present = true;
    self.reaction_1();}
}
main {
  Cockpit c(d, v):();
  Vision  v(d):();
  Door    d():();
}
```

Fig. 7. Timed Rebeca model encoding the semantics of the Lingua Franca model.

In this section, we show how to use Timed Rebeca to verify an engineering model of the aircraft door system, rather than a model mixing engineering requirements and scientific assumptions. Fig. 7 shows a Timed Rebeca model encoding the semantics of the Lingua Franca program shown in Fig. 6. Strictly speaking, this Timed Rebeca model encodes the behavior of the LF program at the *instant* when the Cockpit component schedules the physical action `command`. We discuss the challenge of environmental modeling in Section 6.

Since the LF program is an engineering model, we interpret all reaction dependencies and temporal semantics as *requirements* on the system design.

Therefore, in the Timed Rebeca model, we ensure that the same requirements hold for all message servers that represent LF reactions. In the LF model, when a physical action `command` is scheduled, the `disarm` and `open` input ports of Door should receive messages logically simultaneously, despite the fact that the Vision component sits between the `open` port and the `disarm` port. This temporal requirement necessitates that Vision reaction must be invoked before both reactions in Door, regardless of actual network delays and execution times.

In addition, to preserve determinism, LF requires that all reactions within the same reactor execute in the order of their declarations. This means that when reactions in the Door components are triggered logically simultaneously, the reaction labelled 1 must execute before the reaction labelled 2, as explained in Section 4.1. Therefore, based on the requirements above, LF defines a unique correct sequence of reaction invocations upon the arrival of a physical action:

Cockpit reaction 2 → Vision reaction 1 → Door reaction 1 → Door reaction 2.

A correctly implemented LF runtime must deliver this sequence during program execution, and ensuring that an LF runtime is implemented correctly is nontrivial. When formally verifying the engineering model, however, the LF semantics can be *assumed*, which simplifies verification significantly.

To encode this well defined sequence in Timed Rebeca, we use the `@globalPriority` annotation, which gives a Timed Rebeca message server a priority value respected across reactive classes. These priorities provide partial ordering constraints that, according to Lingua Franca semantics, allow parallel execution while respecting dependencies. This is suitable in our use case, as LF's requirements on the order of reaction invocations spans across reactors. We give the four reactions in the sequence global priorities of 2, 4, 6, and 8 respectively (lines 11, 27, 53, 57 in Fig. 7). We set a reaction's priority to twice its invocation order so that auxiliary handlers (lines 34, 65, 71), which encode the semantics of LF ports, can execute before the reactions they trigger.

We verify this engineering model in Afra, which confirms that `status` can never be 3, even under all possible network latencies and execution times. As long as the implementation correctly follows the LF semantics, we can fully trust this verification result.

6 Challenges

One challenge that arises is that, in order to perform verification, Timed Rebeca programs must usually include a model of the environment in which they operate. A verifiable TR system is closed. All inputs to the designed part of the system have to be modeled. They can be modeled nondeterministically using Timed Rebeca, but in order for bounded model checking to be tractable, the number of nondeterministic choices must be limited. We avoided this problem in the previous section by using a degenerate environment model, where the environment makes exactly one request for service. But for many cases, this will not be sufficient. For example, it may be important to consider the rate at which requests for service can occur. These rates can be bounded in Lingua Franca, which may make the modeling easier, but this remains to be explored.

Since we cannot design the environment in which an engineered system runs, a model of the environment is necessarily a scientific model, not an engineering model. The model can make *assumptions*, enabling a kind of "assume-guarantee" reasoning, but the model cannot be a *specification* of the environment. There is an art, therefore, to constructing models of the environment. Ideally, each such model represents the weakest possible assumptions such that verification remains tractable and the assumptions can reasonably be assumed to hold. In practice, patterns of environment behaviors will need to be verified separately, because any single model with weak enough assumptions to admit all possible behaviors of the environment will lead to intractable verification problems.

7 Conclusion

We observe that models used for verification in practice sometimes conflate engineering models and scientific models, and that there is significant risk that this confusion compromises the trustworthiness of the verification results. We argue that systems under design should be modeled with engineering models, which are specifications of correct behavior, rather than scientific models, which approximate implementations. When this can be done, the results of formal verification can show definitively the correctness of a design. It is a separable problem to determine whether an implementation correctly realizes the specification. A key limitation, however, is that designs can only be verified under assumptions about the environment in which they operate, and that models of the environment are inevitably scientific models, not engineering models.

We have shown that Lingua Franca programs can serve as effective engineering models for a deterministic version of actor networks, and that Timed Rebeca can be used to verify these models. Using Timed Rebeca's ability to model nondeterminism, scientific models of the operating environment can be combined with the engineering models of the design, yielding verification results that are trustworthy under clearly stated assumptions about the operating environment of the designed system.

Acknowledgement. The work in this paper was supported in part by the National Science Foundation (NSF), award #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems) and the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by Denso, Siemens, and Toyota.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Agha, G.: ACTORS: A Model of Concurrent Computation in Distributed Systems. The MIT Press Series in Artificial Intelligence, MIT Press, Cambridge, MA (1986)
2. Agha, G.: Concurrent object-oriented programming. Communications of the ACM **33**(9), 125–140 (1990)
3. Agha, G.A.: Abstracting interaction patterns: A programming paradigm for open distributed systems. In: Stefani, E.N., J.-B. (eds.) Formal Methods for Open Object-based Distributed Systems, IFIP Transactions. pp. 135–153. Chapman and Hall (1997). https://doi.org/10.1007/978-0-387-35082-0_10
4. Bateni, S., Lohstroh, M., Wong, H.S., Kim, H., Lin, S., Menard, C., Lee, E.A.: Risk and mitigation of nondeterminism in distributed cyber-physical systems. In: ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE) (2023). https://doi.org/10.1145/3610579.3613219
5. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., De Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE **91**(1), 64–83 (2003)
6. Box, G.E.P., Draper, N.R.: Empirical Model-Building and Response Surfaces. Wiley Series in Probability and Statistics, Wiley (1987)
7. Cataldo, A., Lee, E.A., Liu, X., Matsikoudis, E., Zheng, H.: A constructive fixed-point theorem and the feedback semantics of timed systems. In: Workshop on Discrete Event Systems (WODES) (2006), http://ptolemy.eecs.berkeley.edu/publications/papers/06/constructive/
8. Hewitt, C.: Viewing control structures as patterns of passing messages. Journal of Artificial Intelligence **8**(3), 323–363 (1977)
9. Khosravi, R., Khamespanah, E., Ghassemi, F., Sirjani, M.: Actors Upgraded for Variability, Adaptability, and Determinism, pp. 226–260. Lecture Notes in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-51060-1_9
10. Kirsch, C.M., Sokolova, A.: The logical execution time paradigm. In: Advances in Real-Time Systems, pp. 103–120. Springer (2012)
11. Lee, E.A.: Determinism. ACM Transactions on Embedded Computing Systems (TECS) **20**(5), 1–34 (July 2021). https://doi.org/10.1145/3453652
12. Lee, E.A., Akella, R., Bateni, S., Lin, S., Lohstroh, M., Menard, C.: Consistency vs. availability in distributed cyber-physical systems. ACM Transactions on Embedded Computing Systems (TECS) **22**(5s), 1–24 (2023). https://doi.org/10.1145/3609119, presented at EMSOFT, September 17-22, 2023, Hamburg, Germany
13. Lee, E.A., Liu, J., Muliadi, L., Zheng, H.: Discrete-event models. In: Ptolemaeus, C. (ed.) System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org, Berkeley, CA (2014), http://ptolemy.org/books/Systems
14. Lee, E.A., Sirjani, M.: What good are models? In: Formal Aspects of Component Software (FACS). vol. LNCS 11222. Springer (2018)
15. Lee, E.A.: Plato and the Nerd — The Creative Partnership of Humans and Technology. MIT Press (2017)
16. Liao, S., Tjiang, S., Gupta, R.: An efficient implementation of reactivity for modeling hardware in the scenic design environment. In: Proceedings of the 34th annual Design Automation Conference. pp. 70–75 (1997)
17. Lohstroh, M., Íncer Romeo, Í., Goens, A., Derler, P., Castrillon, J., Lee, E.A., Sangiovanni-Vincentelli, A.: Reactors: A deterministic model for composable reactive systems. In: 8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy'19). vol. LNCS 11971, p. 27. Springer-Verlag (2019)

18. Lohstroh, M., Lee, E.A., Edwards, S., Broman, D.: Logical time for reactive software. In: Workshop on Timing-Centric Reactive Software (TCRS), in Cyber-Physical Systems and Internet of Things Week (CPSIoT). ACM (2023). https://doi.org/10.1145/3576914.3587494
19. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a lingua franca for deterministic concurrent systems. ACM Transactions on Embedded Computing Systems (TECS) **20**(4), Article 36 (2021). https://doi.org/10.1145/3448128
20. Maler, O., Manna, Z., Pnueli, A.: From timed to hybrid systems. In: Real-Time: Theory and Practice, REX Workshop. pp. 447–484. Springer-Verlag (1992)
21. Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingolfsdottir, A., Sigurdarson, S.H.: Modelling and simulation of asynchronous real-time systems using timed rebeca. Science of Computer Programming **89**, 41–68 (2014). https://doi.org/10.1016/j.scico.2014.01.008
22. Sirjani, M., Jaghoori, M.M.: Ten Years of Analyzing Actors: Rebeca Experience, vol. Lecture Notes in Computer Science, vol 7000. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_3
23. Sirjani, M., Khamespanah, E.: On time actors. Theory and Practice of Formal Methods **LNCS 9660**, 373–392 (2016). https://doi.org/10.1007/978-3-319-30734-3_25, springer, Cham
24. Sirjani, M., Lee, E.A., Khamespanah, E.: Verification of cyberphysical systems. Mathematics **8**, 1–20 (2020). https://doi.org/10.3390/math8071068
25. Zeigler, B.P., Moon, Y., Kim, D., Ball, G.: The DEVS environment for high-performance modeling and simulation. IEEE Computational Science and Engineering **4**(3), 61–71 (1997)