Article

Verification of Cyberphysical Systems

Marjan Sirjani [1]*, Edward A. Lee [2] and Ehsan Khamespanah [3]

[1] School of IDT, Mälardalen University, Sweden; marjan.sirjani@mdh.se
[2] Department of EECS, University of California at Berkeley, USA; eal@berkeley.edu
[3] Department of ECE, University of Tehran, Iran; e.khamespanah@ut.ac.ir
* Correspondence: marjan.sirjani@mdh.se; Tel.: +46-73-662-0517

Version June 6, 2020 submitted to Mathematics

Abstract: The value of verification of cyberphysical systems depends on the relationship between the state of the software and the state of the physical system. This relationship can be complex because of the real-time nature and different timelines of the physical plant, the sensors and actuators, and the software that is almost always concurrent and distributed. In this paper, we study different ways to construct a transition system model for the distributed and concurrent software components of a CPS. We describe a logical-time based transition system model, which is commonly used for verifying programs written in synchronous languages, and derive the conditions under which such a model faithfully reflects physical states. When these conditions are not met (a common situation), a finer-grained event-based transition system model may be required. We propose an approach for formal verification of cyberphysical systems using Lingua Franca, a language designed for programming cyberphysical systems, and Rebeca, an actor-based language designed for model checking distributed event-driven systems. We focus on the cyber part and model a faithful interface to the physical part. Our method relies on the assumption that the alignment of different timelines during the execution of the system is the responsibility of the underlying platforms. We make those assumptions explicit and clear.

Keywords: Cyberphysical systems, Verification, Lingua Franca, Model checking, Rebeca.

---

1. Introduction

Cyberphysical systems (CPSs) are all around us, as in industrial control systems, robotics, smart grids, autonomous cars, and medical devices. Cyberphysical systems are integrations of computation, networking, and physical processes where physical and software components are deeply intertwined. Cyberphysical systems include networked embedded computers monitoring and controlling the physical processes. They also include mechanical, electrical, chemical, or biological components that are controlled or monitored by computer-based algorithms. A study of CPS may emphasize one or the other perspective. Here, we focus on verification of the software controlling the physical processes not the physical processes being controlled by the software.

Formal verification is about assuring properties of models. A holistic approach to verifying CPSs requires models of both the distributed software and physical processes. However, commonly used models for software are incompatible with commonly used models for physical processes [1]. An alternative is to clearly define the interfaces between the cyber and the physical parts of the system and separate the verification problem, from each side relying on the other side to faithfully carry out the semantics of the interfaces. When verifying software, we rely on the hardware to faithfully carry out the operations specified by the software. Hence, when we prove that the software has some property, such as never reaching some undesired state, we can assume that, with high probability, the physical system that executes the software will reflect a corresponding property. The nature of these interfaces, however, and the underlying assumptions they entail become extremely important. For CPS, verification is ultimately about assuring properties of the physical world. This means that it is not sufficient to study the software alone. We need to also study its interactions with its environment.

We use a simple example of a train door controller from Sirjani etal in [2] as our running example. Consider a train door that needs to be locked before the train starts moving. The software controlling train systems is able to lock the door and then send a command to the train to start moving. We can build a model of the software, or write a simple program, and formally verify its correctness. But if we do not know how and when the door gets locked and the train starts moving in response to a software command, then it will do little good to prove that the software never enters a state where it thinks the door is unlocked while the train is moving. The necessity to include the physical aspects of the system, not just its logical ones, is what makes this a CPS.

We can verify that the door's software component is never in the unlocked state while the train's software component is in the moving state. Depending on how the physical interfaces are realized, however, this may or may not align with the physical world. The state of the software system and the state of the physical world are not assured of aligning. What if the door component and the train component are executing on two different microprocessors separated by a network? What does it mean, in this case, for the two to simultaneously be in some state? To have a useful solution we need to address the problem of different timelines in distributed systems and different timelines between the software and the physical world.

A cyberphysical system can be viewed as an interacting pair of reactive systems, one defined in the world of software, and the other in the world of physics. The semantic worlds of physics and software are radically different and often mutually incompatible. So, to prove properties of cyberphysical systems, we may not want to combine models from physics with those of software. Our approach is instead to build a model with a focus on the software side and an abstract (but faithful) model of the physical side. We model the distributed software system that monitors and receives the data from the physical processes and sends the control commands to the physical processes. Using this model we can verify whether, upon receiving certain data, the software system is producing correct output based on the specified requirements. Modeling the input from the physical world "faithfully," and producing "correct" output needs more elaboration. The model of the physical world is reduced to its interface to the software. One major issue in properly modeling this interface is timing.

Time is a critical feature in cyberphysical systems. There is the issue of time in distributed software, and also in the interface of software and the physical world. In order to effectively couple models of software with models of the physical world, we need modeling frameworks that support more than one timeline. We will explain later how we rely on certain assumptions to consider one logical timeline in the model of the software, and how certain guarantees from the selected programming language and the underlying platform allow us to assume that the logical time and the physical time are aligned in such a way that the model of the physical interface stays faithful to the physical world. In other words, our model of the physical inputs is faithful to the physical world and the physical outputs are created correctly. Note that faithfulness and correctness here depends not only on the *values* of the inputs and outputs but also on their timing.

There are alternative approaches in analyzing CPS that are not the topic of this paper (see comprehensive overviews in [3,4]). The focus of the approach can be on modeling the physical processes, the dynamics of the physical quantities. The theory of dynamical control systems is a well-developed discipline rooted in continuous-time models. In a cyberphysical system, the controller consists of discrete software with concurrent components operating in multiple possible modes, interacting with the continuously evolving physical environment. Such systems are often modeled with a mix of finite automata and continuous dynamics, where mode transitions are modeled by discrete, instantaneous state transitions in an automaton, and each state of the automaton is associated with a distinct model of the continuous dynamics. Such models are called hybrid systems [5,6]. We will not consider hybrid system models here. We will instead assume a particular style of software, embodied in the Lingua Franca language, that yields useful and realistic models.

Model checking is a method for formal verification of reactive systems. A model checking tool receives two inputs: a model of the behavior of the system, and a set of properties represented as temporal logic formula showing the desired specification of the same system. A state transition diagram is generated based on the interleaved semantics of the input model and the properties are checked against this state transition diagram. Here we use the Reactive Object Language, Rebeca [7,8], and its model checking tool Afra [9] for formal verification of cyberphysical systems. For doing so, we map Lingua Franca [10–12] programs to an extended version of Timed Rebeca. Lingua Franca is a programming language based on the Reactor model of computation [11] for building cyberphysical systems. Both Rebeca and Lingua Franca are actor-based languages.

The Hewitt Actor model [13,14] is a reference model for concurrent distributed systems with an asynchronous event-driven model of computation. Its event-driven concurrent semantics makes it a natural choice for modeling cyberphysical systems, but it needs to be extended with timing properties. To model the unknown factors in a system, like the possible inputs from the environment, we can use nondeterminism in our Timed Rebeca model. Timed Rebeca [15,16] extends Rebeca to model the timing features. In Timed Rebeca the events are triggered based on their timetag order, and when there is more than one enabled event with the same logical time tag, they are triggered in nondeterministic order. Lingua Franca ensures determinism in a similar way, by first ensuring that messages are handled in timetag order and then also prioritizing reactions within each reactor. To handle simultaneous messages to distinct reactors, LF uses the precedence graph relation between reactors to constrain the order of execution. To model the deteministic behavior of LF, we have extended Timed Rebeca with priorities on message handlers and priorities on actors so that simultaneous messages (those with the same timetag) are handled in a determinstic order. For ordering the execution of actors, priorities are too strong, but they work for the purpose of this paper, which is verification.

There are multiple timelines involved in cyberphysical systems. To have a faithful model of time for cyberphysical systems we need to address both (1) the asynchrony in distributed systems, and (2) the mismatch between physical and logical time. To make analysis possible we need to build layers of abstraction and use assumptions by relying on the other layers. In Timed Rebeca we assume synchronized local clocks for actors that gives us a notion of global time across the model. We use logical timetags, and logical timetags are comparable across all actors in the model. But in distributed systems we cannot assume synchronized clocks for distributed software components, at least not perfect ones. We need certain mechanisms to be able to have such assumption. Ptides[17] and Spanner[18] are two examples that assume synchronized clocks (up to an error bound) and use logical timetags. For distributed actors (as faithful representatives of distributed software components) to be able to have synchronized clocks and comparable timetags we rely on the lower-level network protocols to provide that for us. The second issue is the two timelines of logical world and physical world.

Lingua Franca includes a notion of "logical time" and binds that notion to "physical time" only where the software interacts with the physical world. In Lingua Franca the logical timetag of the input events are assigned based on the physical time of the physical processes. We also need to make sure that the logical timetag of the output events and the physical time of the actuated physical processes have the desired relation. We assume that the actuated outputs affect the physical world within a certain deadline.

Our model stays faithful to the system itself only based on the set of assumptions mentioned above. These assumptions allow us to reason about the system based on the logical timetags in our Rebeca model. Our model may not be a model with the least semantic gap with cyberphysical systems, but we will show in this paper that using model checking we are able to catch many subtle design problems. We show how these problems may exist in very simple examples that exhibit how building such systems can be extremely error-prone. Many of these problems may be related to the timing configurations. We believe that no simple approach exists for verification of cyberphysical systems. Several complimentary methods need to be used to cover the analysis of different aspects of such systems, and in each method we rely on certain assumptions that may be guaranteed by other methods.

In a shorter conference paper version of this paper [19], we raise the interesting questions involved in verification of cyberphysical systems and we used a couple of examples to show how we move towards solving some of the problems. Here we explain the problem and the solutions in a more extensive and structured way using mostly the same examples. The paper is organized as follows. Section 2 introduces the programming model we assume (reactors) and the language in which programs are written (Lingua Franca, LF). It sketches the source code in LF for a running example, a train door controller. Section 3 introduces the Rebeca language and its timed extension, Timed Rebeca, which we further extend here to express temporal properties of Lingua Franca programs. Section 4 explains a translation of the train door running example into this extended Timed Rebeca. Section 5 studies the problem of model checking concurrent LF programs and explains two approaches based on two different semantics with different levels of granularity. Section 6 refines the train door example with programming constructs to control timing and increased interactivity and shows how the Rebeca model checking tool Afra can help identify subtle defects in the design. Section 7 concludes with a discussion of problems that remain open.

2. Lingua Franca and Reactors: Building Cyberphysical Systems

Lingua Franca (LF) [10–12], is a coordination language designed for embedded real-time systems. Software components are called “reactors.” The messages exchanged between reactors have logical timetags drawn from a discrete, totally ordered model of time. Any two messages with the same timetag are logically simultaneous, which means that, for any reactor with these two messages as inputs, if it sees that one message has occurred, then it will also see that the other has occurred. Moreover, every reactor will react to incoming messages in timetag order. If the reactor has reacted to a message with timetag t, no future reaction in the same reactor will see any message with a lesser timetag.

If a reactor produces output messages in reaction to an input, then, by default, the logical time of the output will be identical to the logical time of the input. This principle is borrowed from synchronous languages [20]. The Lingua Franca compiler ensures that all logically simultaneous messages are processed in precedence order, so the computation is deterministic. At a logical instant, the semantics of the program can be given as a unique least fixed point of a monotonic function on a lattice [21], so the computation is deterministic, even if it is distributed across a network. We call this semantics, based on the semantics of synchronous languages, the “logical-time-based semantics.” Here, we also consider an event-based semantics, which becomes useful when an interleaved execution of events with the same logical timetag becomes observable. Event-based semantics has finer granularity compared to logical-time-based semantics.

The syntax of a subset of Lingua Franca is given in Figure 1. The model consists of a set of reactors and a main reactor. Reactors contain state variables, input and output ports, physical actions and reactions. The body of reactions can be written in the target language. As of this writing, LF supports C, C++, and TypeScript. In each case, the LF compiler generates a standalone executable in the target language. A reactor may also react to a “physical action,” which is typically triggered by some external event such as a sensor [11]. The physical action will be assigned a timetag based on the current physical clock on the machine hosting the reactor.

A key semantic property of Lingua Franca is that every reactor reacts to events in timetag order. Preserving this order in a distributed execution is a key challenge. One technique that has proven effective is Ptides [22], a decentralized and fault-tolerant coordination mechanism that relies on synchronized physical clocks with bounded error. The Ptides technique has been applied on a global scale in Google Spanner [23].

Lingua Franca includes a notion of a deadline, which is a relation between logical time and physical time, as measured on a particular platform. Specifically, a program may specify that the invocation of a reaction must occur within some physical-time interval of the logical timestamp of the message. This, together with physical actions, can be used to ensure some measure of alignment between logical time and some measurement of physical time.

$$
\begin{aligned}
\mathit{Model} &::= \mathit{Target}\ \mathit{Reactor}^{*}\ \mathit{MainReactor}\\
\mathit{Target} &::= \textbf{target}\ \mathit{targetLanguageName}\textbf{;}\\
\mathit{Reactor} &::= \textbf{reactor}\ \mathit{reactorName}\ \{\ \mathit{StateVar}^{*}\ \mathit{Input}^{*}\ \mathit{Output}^{*}\ \mathit{Action}^{*}\ \mathit{Reaction}^{*}\}\\
\mathit{StateVar} &::= \textbf{state}\ \mathit{varId}\ \textbf{:}\ \mathit{typeId}\ (\mathit{initialValue});\\
\mathit{Input} &::= \textbf{input}\ \mathit{inputId}\ \textbf{:}\ \mathit{typeId}\textbf{;}\\
\mathit{Output} &::= \textbf{output}\ \mathit{outputId}\ \textbf{:}\ \mathit{typeId}\textbf{;}\\
\mathit{Action} &::= \textbf{physical action}\ \mathit{actionId}\ \textbf{:}\ \mathit{typeId}\textbf{;}\\
\mathit{Reaction} &::= \textbf{reaction}\ (\mathit{Trigger}^{*})\ [\textbf{->}\ \mathit{outputId}(,\ \mathit{outputId})^{*}]\ \textbf{\{=}\ \mathit{Code}^{*}\textbf{=\}}\\
\mathit{Trigger} &::= \mathit{inputId} \mid \mathit{actionId}\\
\mathit{Code} &::= \mathit{Target} - \mathit{Language} - \mathit{Statement}\\
\mathit{MainReactor} &::= \textbf{main reactor}\ \mathit{mainReactorName}\ \textbf{\{}\ \mathit{Instantiation}^{*}\ \mathit{Connection}^{*}\ \textbf{\}}\\
\mathit{Instantiation} &::= \mathit{id} = \textbf{new}\ \mathit{reactorName}()\ \textbf{;}\\
\mathit{Connection} &::= \mathit{id.inputId}\ \textbf{->}\ \mathit{id.outputId}\ [\textbf{after}\ \mathit{delayValue}]\textbf{;}
\end{aligned}
$$

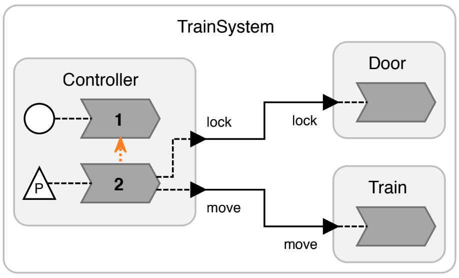
Figure 1. Syntax of a subset of Lingua Franca that we use in our examples in this paper (adapted from Lingua Franca Github [24]). The syntax is written in a slightly revised version of Extended BNF where instead of putting terminals in quotations we use words in "**bold**" format. Angled brackets ⟨...⟩ are used as meta parenthesis, superscript $+$ for repetition at least once, superscript $*$ for repetition zero or more times, whereas using ⟨...⟩ with repetition denotes a comma separated list. Brackets [...] indicates that the text within the brackets is optional. In the syntax, *targetLanguageName*, *reactorName*, *mainReactorName* stand for the target language for LF, the name of the reactor, and the name of the main reactor, respectively. The *varId*, *typeId*, *inputId*, *outputId*, *actionId* stand for the names of a variable, a type, an input and an output, respectively; and `id` stands for the name of an instance of a reactor. *Target-Language-Statement* stands for the statements of the target language. In the *Reactor* rule the components do not need to come in the presented order.

2.1. The Simple Train Door Controller in Lingua Franca

Consider a train door that needs to be locked before the train starts moving. The software controlling train systems is able to lock the door and then send a command to the train to start moving. Consider in Figure 2b the sketch of an implementation of a highly simplified version of such train controller software in Lingua Franca. In this use, the code shown in the figure gets translated into C code that can run on a train's microcontrollers. Similar realizations could be built in any of a number of model-based design languages, including any of the synchronous languages [20] (SCADE, Esterel, Lustre, SIGNAL, etc.), Simulink, LabVIEW, ModHel'X [26], Ptolemy II [27], or ForSyDe [28], to name a few. All will raise similar issues to those we address in this paper.

The structure of the code is illustrated in Figure 2a. It consists of three components called "reactors," instances of the reactor classes Controller, Door, and Train. The main reactor (starting on line 30) instantiates and connects these components so that the controller sends a messages to both the door and the train. These components could be implemented on a single core, on multiple cores, or on separate processors connected via a network.

Let's focus first on the interaction between these components and the physical world. The Controller reactor class defines a `physical action` named "`external_move`" (line 5), which in Lingua



(a) Structure of the simple door controller example. This image is rendered automatically by the Lingua Franca IDE using the KIELER Lightweight Diagrams framework [25].

```
1  target C;
2  reactor Controller {
3    output lock:bool;
4    output move:bool;
5    physical action external:bool;
6    reaction(startup) {=
7      ... Set up sensing.
8    =}
9    reaction(external)->lock, move {=
10     set(lock, external_value);
11     set(move, external_value);
12   =}
13 }
14 reactor Train {
15   input move:bool;
16   state moving:bool(false);
17   reaction(move) {=
18     ... actuate to move or stop
19     self->moving = move;
20   =}
21 }
22 reactor Door {
23   input lock:bool;
24   state locked:bool(false);
25   reaction(lock) {=
26     ... Actuate to lock or unlock door.
27     self->locked = lock;
28   =}
29 }
30 main reactor System {
31   controller = new Controller();
32   door = new Door();
33   train = new Train();
34   controller.lock -> door.lock;
35   controller.move -> train.move;
36 }
```

(b) Lingua Franca code for the simple door controller example in Figure 2a with a potential defect.

```
reactiveclass Controller(5) {
  knownrebecs {
    Door door;
    Train train;
  }
  statevars { boolean moveP; }
  Controller() {
    self.external();
  }
  msgsrv external() {
    boolean oldMoveP = moveP;
    moveP = ?(true,false);
    if(moveP != oldMoveP) {
      door.lock(moveP);
      train.move(moveP);
    }
    self.external() after(1);
  }
}
reactiveclass Train(5) {
  statevars { boolean moving; }
  Train() {
    moving = false;
  }
  msgsrv move(boolean tmove) {
    if (tmove) {
      moving = true;
    } else {
      moving = false;
    }
  }
}
reactiveclass Door(5) {
  statevars { boolean is_locked; }
  Door() {
    is_locked = false;
  }
  msgsrv lock (boolean lockPar) {
    is_locked = lockPar;
  }
}
main {
  @priority(1) Controller controller(door,
                                     train):();
  @priority(2) Train train():();
  @priority(2) Door door():();
}
```

(c) Timed Rebeca model (extended with priorities) for the simple door controller example in Figure 2a.

Figure 2. The structure, Lingua Franca program, and Timed Rebeca model for the simple door controller example.

Franca is an event that is triggered by something outside the software system and is then assigned a logical timetag that approximates the physical time at which that something occurred in the physical world [11]. In practice, in the `reaction(startup)` block of code (starting on line 6), which executes upon startup of the system, the reactor could set up an interrupt service routine (ISR) to be invoked whenever the driver pushes a button to make the door lock and train move. The ISR would call an LF function `schedule` to trigger the action and assign it a timetag. The `reaction` to the `external_move` action (starting on line 9) will be invoked when logical time reaches the assigned timetag. This reaction sets the outputs named "`lock`" and "`move`" to the Boolean value true. Since that outputs are connected to the input named "`lock`" of the door component (line 34) and the input named "`move`" of the train component (line 34), respectively, this results in a message to the door component and a message to the train component at the logical time of the timetag.

The train component has a state variable named "`moving`" (line 16) that changes value when it receives a message on its "`move`" input port (line 19). The variable has value true when the train is moving and false when the train is stopped. The door component has a state variable named "`locked`" (line 24) that changes value when it receives a message on its "`lock`" input port (lines 23 and 27).

3. Timed Rebeca: Model Checking Cyberphysical Systems

The Reactive Object Language, Rebeca [7,8], is an actor-based [13,14] modeling language supported by a model checking tool Afra [9]. Rebeca is used for modeling and formal verification of concurrent and distributed systems. The model of computation in Rebeca is event-driven and the communication is asynchronous. The grammar is shown in Figure 3. Actors have message queues; each actor takes the message on the top of the queue, executes the method related to that message (called message server) in an atomic and non-preemptive way. While executing a method, messages can be sent to other actors (or itself), and the values of the state variables can change. Sending messages is non-blocking, and there is no explicit receive statement.

In Timed Rebeca [15,29,30] three keywords are added to model logical time: `delay`, `after` and `deadline`. Timetags are attached to messages and states of each actor. Here we have a buffer of timetagged messages instead of a message queue. Using the keyword `delay`, one can model progress of time while executing a method. If a `send` statement is augmented by `after(t)`, the timetag of the message when it is put in the queue of the receiver is `t` units more than the timetag of the message when it is sent. The timetag of the message when it is sent is the current logical time of the sender. By using `after`, one can model the network delay; periodic events can be modeled using send messages to itself augmented by `after`. The `deadline` keyword models the timeout; if the current time of the receiver actor at the time of triggering the event (taking the message to handle it) is more than the expressed deadline then the model checking tool will complain and raise the deadline-miss warning. While mapping Lingua Franca programs to Timed Rebeca we only use the `after` construct and it is used to increase the value of the logical timetag of the message, like in LF.

The original Rebeca language does not have a model of time and handles incoming messages in nondeterministic order. Timed Rebeca adds a model of time, but still handles incoming messages at each logical time in nondeterministic order. Our extension supports annotating Rebeca actors, and also their message servers, with priorities. These priorities can enforce the ordering constraints on message handlers that are defined by the Lingua Franca language.

The external physical inputs in Lingua Franca are modeled as sending those messages to self. These messages are sent to self augmented with the `after` construct. We can assign nondeterministic values to `after` and hence the messages are received some time later nondeterministically. Of course we can also model a periodic physical input by assigning the period of arrival as the value of the `after` construct.

3.1. A Simple Train Door Controller in Timed Rebeca

A (slightly simplified) Timed Rebeca model of the program in Figure 2b is shown in Figure 2c. Given this model, we can use the Afra model checking tool to get the transition system model and to check safety properties. An interesting point in this Rebeca code is modeling the production of the stimulus that triggers reactions. We need to model the environment or the interface to the physical world. On line 8, the constructor for the Controller sends itself the message `external`. On line 12 in the `external` method the value of `moveP` is set to `true` or `false` nondeterministically to show the possibility of presence or absence of the external message. This is how we model possible external

$$
\begin{aligned}
\mathit{Model} &::= \mathit{Class}^{*}\ \mathit{Main} \\
\mathit{Class} &::= \textbf{reactiveclass}\ \mathit{className}\ (\mathit{queueLength})\ \{\ \mathit{KnownRebecs}\ \mathit{Vars}\ \mathit{Constructor}\ \mathit{MsgSrv}^{*}\ \} \\
\mathit{KnownRebecs} &::= \textbf{knownrebecs}\ \{\ \mathit{VarDcl}^{*}\ \} \\
\mathit{Vars} &::= \textbf{statevars}\ \{\ \mathit{VarDcl}^{*}\ \} \\
\mathit{VarDcl} &::= \mathit{type}\ \langle v \rangle^{+}; \\
\mathit{Constructor} &::= \mathit{className}\ (\langle \mathit{type}\ v \rangle^{*})\ \{\ \mathit{Stmt}^{*}\ \} \\
\mathit{MsgSrv} &::= \textbf{msgsrv}\ \mathit{methodName}(\langle \mathit{type}\ v \rangle^{*})\ \{\ \mathit{Stmt}^{*}\ \} \\
\mathit{Stmt} &::= v{=}e;\ |\ v{=}?(e\langle ,e \rangle^{+});\ |\ \mathit{Call};|\ \textbf{if}\ (e)\ \{\ \mathit{Stmt}^{*}\ \}\ [\textbf{else}\ \{\ \mathit{Stmt}^{*}\ \}];\ |\ \textbf{delay}(t); \\
\mathit{Call} &::= \mathit{rebecName}.\mathit{methodName}(\langle e \rangle^{*})\ [\textbf{after}(t)][\textbf{deadline}(t)] \\
\mathit{Main} &::= \textbf{main}\ \{\ \mathit{InstanceDcl}^{*}\ \} \\
\mathit{InstanceDcl} &::= \mathit{className}\ \mathit{rebecName}\ (\langle \mathit{rebecName} \rangle^{*}){:}(\langle \mathit{literal} \rangle^{*});
\end{aligned}
$$

Figure 3. Syntax of Timed Rebeca (adapted from [30]). The notation is the same as that in Figure 1. Identifiers *className*, *rebecName*, *methodName*, *queueLength*, v, *literal*, and *type* denote class name, rebec name, method name, queue length, variable, literal, and type, respectively; and e denotes an (arithmetic, boolean or nondetermistic choice) expression. In the instance declaration (rule *InstanceDcl*), the list of rebec names ($\langle \mathit{rebecName} \rangle^{*}$) passed as parameters denotes the known rebecs of that instance, and the list of literals ($\langle \mathit{literal} \rangle^{*}$) denotes the parameters of its constructor.

stimulus at different times. If this value is changed from the previous period (comparing `moveP` and `oldMoveP` on line 13) then the two message servers `lock` and `move` are called to lock (or unlock) the door and move (or stop) the train (lines 14 and 15). This `external` message is sent to itself every one time unit by the controller (line 17).

4. Mapping of Reactors to Timed Rebeca with Priorities

Table 1 shows the mapping between Reactors and Timed Rebeca (extended with priorities). Each reactor in Lingua Franca is mapped to a reactive class, and each reaction is mapped to a message server in Rebeca. The trigger in a reaction is the name of the message server, and states in LF are mapped to state variables in Rebeca. Rebeca is an object-based language, not a component-based one. Actors call each other instead of writing on a port. In Lingua Franca we build the bindings between inputs and outputs explicitly in the connection part of the program. In LF a reaction reacts to a trigger, and the trigger is one of the inputs to the reactor. A reaction has outputs and those outputs are set by assigning values to them. Then in the connection part of the main reactor, all the bindings are set by defining which input of which reactor is connected to which output of which reactor. This way the flow of data is realised. You can change the topology by changing the connections. In Rebeca, a message server of other rebecs (or self) is called, and that is how the binding and the flow is realised. There is also a list of known rebecs in a reactive class that shows the rebecs to whom you may send messages to.

For the timing issues, there is an after keyword in Lingua Franca that has the same semantics as in Timed Rebeca. The timetag of the sent message is increased by the value of the after. Rebeca has a delay construct which is not used in LF. Delay in Rebeca increases the timetag within a message server. This has no use in synchronous languages.

In Lingua Franca the messages are handled in timetag order, for the messages with the same timetag the reactions are prioritized within each reactor. To handle simultaneous messages to distinct reactors, LF uses the precedence graph relation between reactor to constrain the order of execution. To faithfully model the LF programs,Timed Rebeca is extended with priorities. The pririties are added by annotations to both message servers and rebecs. The precedence graphs in LF cannot necessarily be mapped into priorities, but priorities are enough for the purpose of this paper. Adding the information of precedence graphs to a Rebeca model in the main can be done with no difficulty and is considered as a future work.

We can also describe part of the mapping using the structure diagram in Figure 2a. The triangle with the "P" is the physical action in LF and external message server in Rebeca, the circle is the "startup" event in LF and the message sent in the constructor message server in Rebeca, the V-shape arrows are reactions in LF and message servers in Rebeca, and the red arrows between the reactions (message servers) are dependencies in LF, and priorities in Rebeca.

The mapping between Reactors and Timed Rebeca is natural and can easily be done. In Lingua Franca we can write the body of reactions in any target language that LF supports. In this work we write the body of reactions in Rebeca. After the code is model checked and debugged, then the Rebeca code needs to be translated to one of the languages supported by LF to be able to execute the LF program. Many design problems can be revealed by model checking the abstract model when the complicated target code is not yet in place. We can also consider mapping the target codes to Rebeca, that is left for the future work.

Lingua Franca Construct/Features	**Timed Rebeca Construct/Features**
reactor	*reactiveclass*
reaction	*msgsrv*
trigger	*msgsrv name*
state	*statevars*
input	*msgsrv*
output	*known rebecs*
physical action	*msgsrv*
implicit in the topology	*Priority*
main	*main*
instantiation ***(new)***	*instantiation of rebecs*
connection	*implicit in calling message servers*
after	*after*
–	*delay*

Table 1. The mapping between Lingua Franca and Timed Rebeca

5. Logical-time-based and Event-based Semantics

A transition system model, which is needed for model checking, requires a concept of the "state" of a system at a particular "instant in time." It does not require that "time" be Newtonian time, measured in seconds, minutes, and hours and aligned to the Earth's orbit around the sun. Instead, it only requires a concept of simultaneity, where the "state" of the system is the composition of the states of its components at a "simultaneous instant," whatever that means in the model. In Lingua Franca, we can define a "simultaneous instant" to be the endpoint when all reactions at a logical time have completed. The "state" at that "instant" can be defined to be the combination of the state variable valuations of all the reactors at that "instant." This is the approach commonly used in synchronous languages, where transient states during the computation at a logical time are ignored. We call this interpretation a **logical-time-based semantics**.

To perform verification formally, we need to build a state-transition model of the program. Figure 4b gives the logical-time-based semantics of the program in Figure 2b. In the initial state, the door is unlocked and the train is not moving. This state transition system shows that at each logical time, the program will nondeterministically either remain in the same state (indicated by the self-loop transitions) or change to the other state. Once the program is in the new state, at subsequent logical times, it will similarly nondeterministically remain in the same state or transition back to the initial state. This transformation relies on the semantics of Lingua Franca being rooted in the fixed-point semantics of synchronous languages [21].

Looking at Figure 4b, it is obvious that the model never enters a state where the train is moving and the door is unlocked. The transition system model is so simple in this case that there is no need for a model checker to verify this property.

This approach to verification is sound because it accurately and correctly models the semantics of the program. But the astute reader should be nervous. What if the door component and the train component are executing on two different microprocessors separated by a network? In this case, there will be a physical time delay between when the train begins moving and the door gets locked, even if there is no logical time delay. In this case, the verification exercise is simply misleading unless we consider this delay in our model.

In the Lingua Franca software, the offending physical state of the system, where the train is moving and door is unlocked, is a transitory state occupied briefly during the computation at a logical time instant. Its duration in logical time is exactly zero. If the physical system is designed in such a way that the physical environment can only observe states with non-zero logical time duration, then we can have confidence in the safety conclusion.

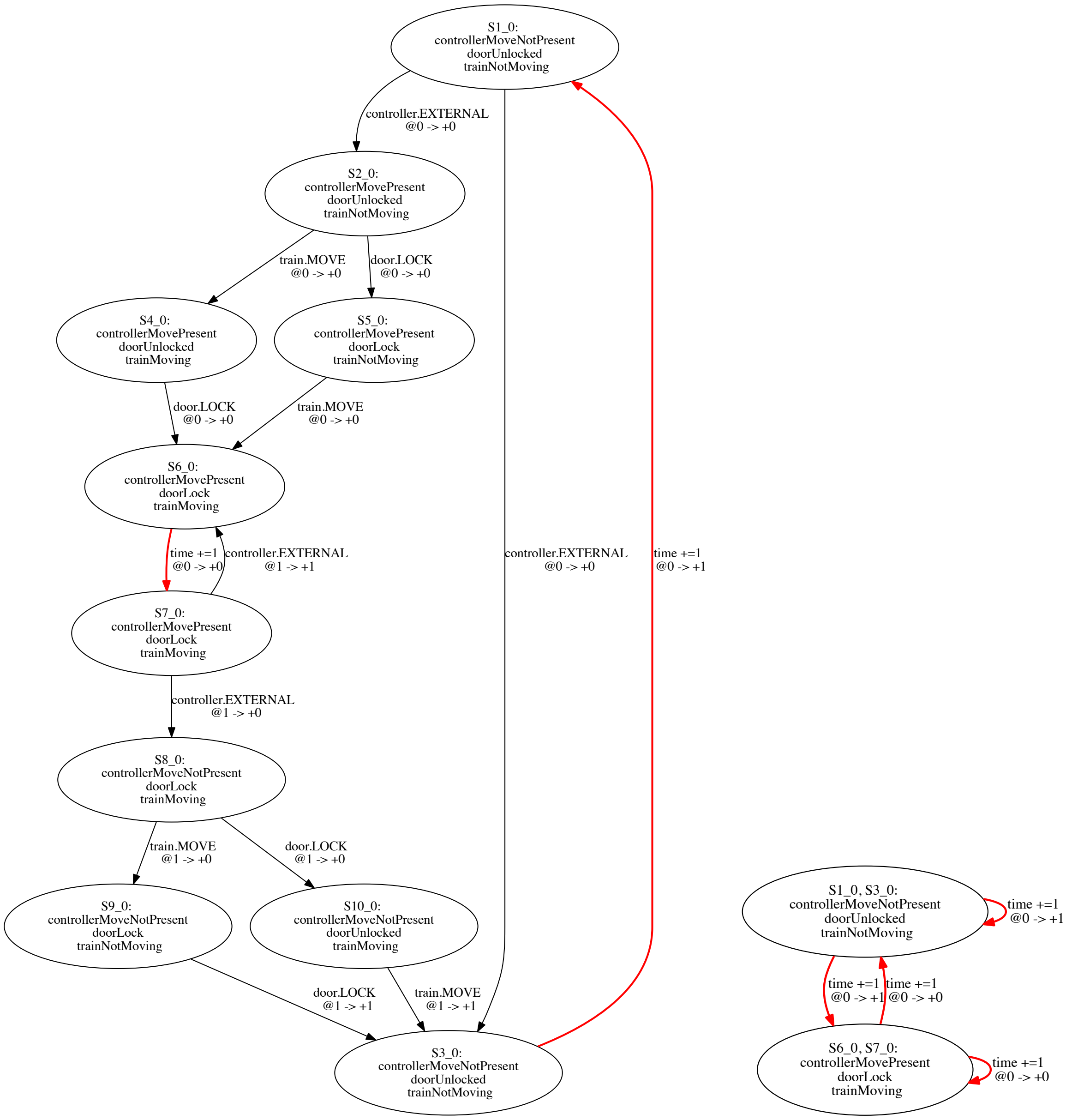
It is not uncommon to design control system hardware precisely to make such guarantees. Programmable Logic Controllers (PLCs), which are widely used to control machinery in industrial automation, have mechanisms that provide such guarantees [31,32]. In particular, PLC software does not directly interact with physical actuators. Instead, during a cycle of execution, the software components write commands to a buffer in memory, and only after the cycle is complete does the hardware read from that memory and drive the physical actuators. If the memory goes through transitory unsafe states during the execution of a cycle, those unsafe states are guaranteed to have no effect on the physical world. If Lingua Franca were to be deployed on hardware with such an I/O system, where a "cycle" is defined by the completion of all reactions at a logical time, then no safety violation would occur. However, this conclusion is not based on the program alone, but rather on a deep and tricky analysis of the program and the hardware on which it is executing. Moreover, the PLC-style semantics is difficult to realize on a distributed system. If the Door component and the Train component are executing on distinct microprocessors, then ensuring that their actuations occur only after a logical-time cycles has been completed requires fairly sophisticated distributed control over the program execution. Perhaps a better approach is to model the steps in the execution in more detail and attempt to design the program to be safe even without such a sophisticated I/O system. We will do that next.

A Lingua Franca execution can be modeled as a sequence of reaction invocations, where each reaction is atomic. We call such a model an **event-based semantics**. It is more fine grained than the logical-time-based semantics and it includes a sequence of steps performed during a logical time instant. Each step is one invocation of a reaction in the Lingua Franca program. Each reaction is triggered by one or more "events," where an "event" is either a message sent between components or an action that has been scheduled by a call to the **schedule** function in Lingua Franca. Every such event occurs at a logical time instant.

5.1. The State Transition Diagram and the Safety Property of the Example

For this simple system, the safety property of interest is that the door be locked while the train is moving. This can be posed as a formal verification problem, where the goal is to prove this property. In the program shown in Figure 2b, the door and train components have state variables, and we can attempt to verify that the door is never in the unlocked state while the train is in the moving state. Depending on how the physical interfaces are realized, however, this may or may not align with the physical world. We can use the features of Lingua Franca to assure that the state of the software system and the state of the physical world are aligned.

We can use Afra model checking tool to get the event-based state transition system of the Rebeca model in Figure2c and to check safety properties. The event-based transition system is shown in Figure 4a. The transitions shown in black in Figure 4a are transitions that all occur at the same logical time.



(a) The Event-based transition system model which is generated by Afra [9]

(b) The Logical-time-based semantics of the model

Figure 4. Transition system model of the Timed Rebeca model in Figure 2c

The transitions shown in red coincide with the advancement of logical time. Thus, Figure 4b can be understood to be an abstraction of this transition diagram that aggregates all the intermediate states at each logical time into one single state. The self-loops in Figure 4b are represented as the transitions from S6_0 to S7_0 and back, and S1_0 to S3_0 and back in Figure 4a.

The transition system of Figure 4a is a slightly revised version of the transition system generated automatically by Afra. In this transition system, the state labeled "S4_0" violates our safety requirement. The train is moving and the door is unlocked. There is a safe trace, going through S5_0 instead of S4_0, but the interleaving semantics allows either trace. Similarly, the state labeled "S10_0" is also not safe. Here we see the so-called diamond effect that is well-known in the model checking domain and may be created when two transitions are enabled in the same state (like in states "S2_0" and "S8_0") and are chosen nondeterministically. If the I/O system makes these transitory states invisible to the environment, as could be done using the PLC style of I/O, then we do not need this finer grained transition system model and could instead have verified the safety property using the much simpler logical-time-based model of Figure 4b. Without such an I/O system, however, we have more work to do before we can have confidence in this system.

6. Extending the Simple Train Door Controller

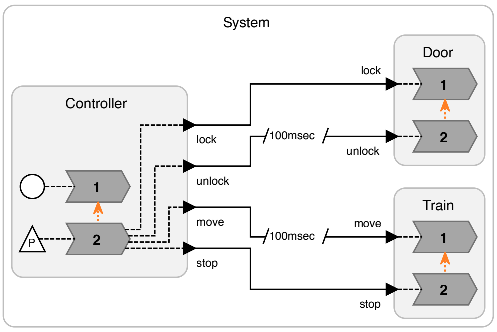
We use variations of a simple train door example from Sirjani etal. [2] to show how we address different questions raised in verification of CPS. In Figure 2, we show a model with three components, a train, an external door in the train, and a controller that commands the door to lock, and the train to move. In Section 6.1, we add timing features to the example and show how we can fix the problem of program in Figure 2 by the proper timing features. We also show the subtleties with the timing and how we can easily make design mistakes. We also show how the external physical triggers can put the system at risk and jeopardize the safety property. In Section 6.2, we show an example where an external physical action can block the progress of system. For example a passenger can keep pressing the open door button and hence stopping the door from being closed and locked, and as a consequence prevent the departure of the train.

6.1. The Train Door Controller with Timing Features

The flaw identified by the Afra tool in the program in Figure 2b can be corrected with a slightly more sophisticated Lingua Franca program. Note that the flaw only exists if we consider the event-based semantics of the program. A simple way is to define two reactions of move and stop in the train reactor (instead of just one reaction of move that decides to actuate move or stop based on the input parameter), and lock and unlock in the door reactor (instead of just one reaction of lock that decides to actuate lock or unlock based on the input parameter), and increment the timetag of an unlock or move message so that it has a logical timetag that is strictly larger than the corresponding stop or lock message. Such a Lingua Franca program is shown in Figure 5b. It has the structure shown in Figure 5a.

Here, we use the `after` keyword on lines 44 and 45 to increment the timetag of the messages by a specified amount (100 msec). This keyword has exactly the same semantics in Lingua Franca and Timed Rebeca, so it creates no complications in translation. With these changes, when the Controller requests that the train move, it issues a lock message with the timetag of the original request and a move message with a timetag incremented by 100 msec. When it requests that the train stop, the unlock message is similarly delayed. This change required separating the lock from the unlock signal and the move from the stop signal because the logical time properties of these pairs of signals differ. In Figure 2a, by contrast, lock and unlock are carried by a single Boolean, as are move and stop.

We can adjust the Timed Rebeca model to match this new design (see Figure 5c) and re-run the model checker. This time, Afra reveals a more subtle problem that can occur if the system has no constraints on the spacing between timetags of successive `external` events. Suppose that the train is stopped and the door is unlocked and we received `external = true` at logical time 0. This will result



(a) Structure of the door controller example

```
target C;
reactor Controller {
  output lock:bool; output unlock:bool;
  output move:bool; output stop:bool;
  physical action external:bool;
  reaction(startup) {=
    ... Set up external sensing.
  =}
  reaction(external)
      ->lock, unlock, move, stop {=
    if (external_value) {
      set(lock, true); set(move, true);
    } else {
      set(unlock, true); set(stop, true);
    }
  =}
}
reactor Train {
  input move:bool; input stop:bool;
  state moving:bool(false);
  reaction(move) {=
    self->moving = true;
  =}
  reaction(stop) {=
    self->moving = false;
  =}
}
reactor Door {
  input lock:bool; input unlock:bool;
  state locked:bool(false);
  reaction(lock) {=
    ... Actuate to lock door.
    self->locked = true;
  =}
  reaction(unlock) {=
    ... Actuate to unlock door.
    self->locked = false;
  =}
}
main reactor System {
  c = new Controller(); d = new Door();
  t = new Train();
  c.lock -> d.lock;
  c.unlock -> d.unlock after 100 msec;
  c.move -> t.move after 100 msec;
  c.stop -> t.stop;
}
```

(b) Variant of Figure 2b that manipulates timetags. The values for **after** are set to 100.

```
reactiveclass Controller(5) {
    knownrebecs{
             Door door; Train train;
    }
    statevars { boolean moveP;}
    Controller() {
       moveP = true;
       self.external_move();
    }
    msgsrv external_move() {
      int d = ?(0, 50);    // lock, stop
      int x = ?(51, 99);  // move, unlock
      int extd = 100;      // external_move
      if (moveP) {
           door.lock() after(d);
           train.move() after(x);
        } else {
          door.unlock() after(x);
          train.stop() after(d);
        }
        moveP = !moveP;
        self.external_move() after(extd);
  }  }
 reactiveclass Train(10) {
    statevars{
      boolean moving;
      }
    Train() {
       moving = false;
       }
    @priority(1) msgsrv stop() {
       moving = false;
    }
    @priority(2) msgsrv move() {
       moving = true;
}  }
reactiveclass Door(10) {
    statevars{
      boolean is_locked;
    }
     Door() {
       is_locked = false;
     }
    @priority(1) msgsrv lock () {
        is_locked = true;
    }
    @priority(2) msgsrv unlock () {
        is_locked = false;
    }
}
main {
     @priority(1) Controller controller(door,
                        train):();
     @priority(2) Train train():();
     @priority(2) Door door():();
}
```

(c) The Rebeca model for the code in Figure 5b. Note that here we put different values for **after** constructs comparing to the LF code in Figure 5b.

Figure 5. The Train Door Example with delays in lock/unlock and stop/move

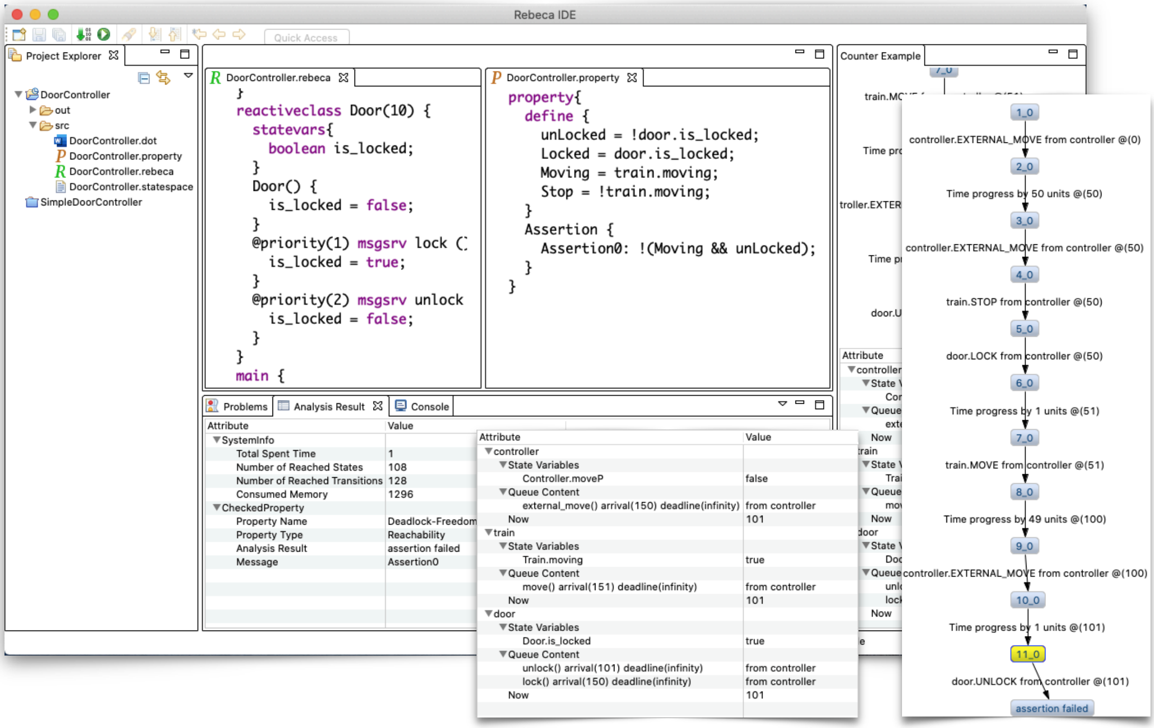


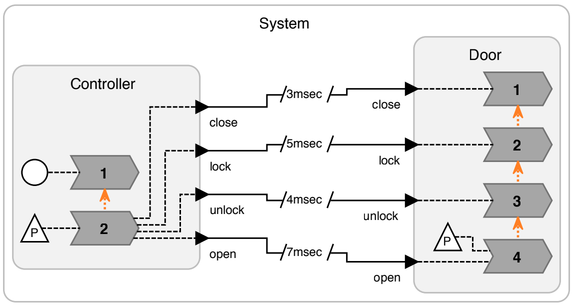
Figure 6. An snapshot of Afra finding the counterexample for the model in Figure 5c where the physical external event occurs every 50 units of time (variable `extd` in the model is set to 50). The value of the variables and the contents of the message buffer in state 11_0 is shown in the snapshot.

in a `lock` message to the Door with timetag 0 and a `move` message to the Train with timetag 100 msec. Suppose that we then receive `external = false` at logical time 50 msec. This will result in a `stop` message to the Train with timetag 50 msec, overtaking the `move` message! But worse, it will send an `unlock` message with timetag 150 msec, and the door will unlock while the train is moving! This new flaw is revealed by a counterexample generated by Afra shown in Figure 6. In Figure 6, you may see the Afra interface with the Rebeca model and the property file including the assertions. The Boolean variables defined in the “define” part of the property file are those that are shown in the states in the transition system. In the right corner of the figure, the trace of the counterexample is shown and the values of variables in state 11_0 (the state right before the assertion is failed) are shown in another window.

This new flaw is not correctable by simply manipulating logical timetags. The flaw pertains to the relationship between physical time and logical time (having no constraints on the spacing between timetags of successive `external` events that represent physical actions), and our verification strategy here stays entirely in the world of logical time. A similarly cross-cutting flaw could occur if the later timetag of the `move` event does not result in a later occurrence of the train moving physically. Again, this flaw pertains to the relationship between physical and logical times, a relationship that is ultimately established not only by the software in the systems, but rather by the combination of software and hardware.

6.2. The Train Door Controller and a Passenger

Another example is shown in Figure 7. In this example we only show the controller and the door. Here the door accepts four commands of unlock, open, close and lock. When the train stops at a platform the controller unlock and then opens the door. When the train is ready to move the controller first close and then lock the door. The train can only start moving if the doors are locked. But the



(a) Structure of the door controller example

```
target C;
reactor Controller {
  output lock:bool; output unlock:bool;
  output open:bool; output close:bool;
  physical action external:bool;
  reaction(startup) {=
    // ... Set up external sensing.
  =}
  reaction(external)->close, lock, open,
                      unlock {=
    if (external_value) {
      set(close, true); set(lock, true);
    } else {
      set(open, true); set(unlock, true);
    }
  =}
}
reactor Door {
  input lock:bool; input unlock:bool;
  input open:bool; input close:bool;
  physical action ext_open:bool;
  state locked:bool(false);
  state is_open:bool(false);
  reaction(close) {=
    ... Actuate to close door.
    self->is_open = false;
  =}
  reaction(lock) {=
    ... Actuate to lock door.
    if(!self->is_open)
       self->locked = true;
  =}
  reaction(unlock) {=
    ... Actuate to unlock door.
    self->locked = false;
  =}
  reaction(open, ext_open) {=
    ... Actuate to open door.
    if(!self->locked)
       self->is_open = true;
  =}
}
main reactor System {
  c = new Controller();
  d = new Door();
  c.lock -> d.lock after 5 msec;
  c.unlock -> d.unlock after 4 msec;
  c.open -> d.open after 7 msec;
  c.close -> d.close after 3 msec;
}
```

(b) Variant of Figure 5b with open door.

```
reactiveclass Controller(5) {
  knownrebecs{
    Door door;
  }
  statevars {
    boolean moveP;
  }
  Controller() {
    moveP = true;
    self.external_move();
  }
  msgsrv external_move() {
    int closingDelay = 3;
    int lockingDelay = 5;
    int unlockingDelay = 4;
    int openingDelay = 7;
    if (moveP) {
      door.close() after(closingDelay);
      door.lock() after(lockingDelay);
    } else {
      door.unlock() after(unlockingDelay);
      door.open() after(openingDelay);
    }
    moveP = !moveP;
    self.external_move() after(10);
  }
}
reactiveclass Door(10) {
  statevars{ boolean is_locked, is_open; }
  Door() {
    is_locked = false; is_open = true;
    self.external_open() after(1);
  }
  @priority(1) msgsrv close() {
    is_open = false;
  }
  @priority(2) msgsrv lock (){
    if(!is_open)
      is_locked = true;
  }
  @priority(3) msgsrv unlock() {
    is_locked = false;
  }
  @priority(4) msgsrv open() {
    if(!is_locked)
      is_open = true;
  }
  @priority(1) msgsrv external_open() {
    int retryDelay = 2;
    self.open();
    self.external_open() after(retryDelay);
  }
}
main {
  @priority(1) Controller controller(door):();
  @priority(2) Door door():();
}
```

(c) The Rebeca model for the code in Figure 7b.

Figure 7. The Door Control Example with an external passenger pressing the **open** button. Here we added **open** and **close** commands for the door and do not show the train.

door can only be locked if the door is closed. Here we assume that there is an open button for the passengers also. So, a passenger can press the open button before the door is locked. The controller sends the close and then the lock command, but there is a scenario in which the external open button is pressed quickly enough that the door can never be locked, and consequently the train can never move.

We can change the values of the after construct for different commands and see various behaviors of the system for different configurations. With the configuration shown in Figure 7 the door will never be locked. We checked the assertion of door being unlocked and it is satisfied in this scenario showing that the door stays unlock all the time. If we increase the value of after in sending `external_open` it will eventually be large enough to allow the door to close and then lock.

7. Discussion and Future Work

The combination of a language like Lingua Franca with an explicit model of time, and a model checking tool like Timed Rebeca with Afra can prove quite effective for finding a number of bugs. Although Rebeca language, which is similar to Java, is expressive enough, it is not clear whether it would be accepted by designers as a target language, and the toolchains do not currently exist to compile it down to code that could execute in microcontrollers as would be needed to deploy the train controller. If these toolchains are created, however, the result could be a very effective package for designing and deploying formally verifiable CPS software. However, there are some serious limitations that warrant further research.

Based on our (limited number of) experiments and our insights, the mapping between Lingua Franca and Timed Rebeca can be simple as long as we stay in the logical time domain of Lingua Franca (and as long as the reaction code in Lingua Franca can be translated to message server code in Timed Rebeca). By mapping LF to Rebeca and through our examples we demonstrated a set of problems that can be found using model checking. In the first example we show the model and how logical-time-base semantics and event-based semantics are different, and how we rely on the underlying platform to guarantee that the observable behavior is base on the for example logical-time-base semantics. When we have distributed CPS this may become a nonrealistic assumption. The second example shows how timing comes in, and how we can rely on different timing configurations to build correct cyberphysical systems. It also shows how subtle problems may raise that are not easy for a designer to notice, and how model checking helps in revealing such problems. In the last two examples we show the connection between the logical and the physical timelines, and the effect of physical events on the logical behavior of the software and how by using model checking we can move towards finding such problems.

Because Rebeca is designed for model checking, Rebeca models are closed, meaning that there are no external inputs. The reactions that can be triggered from outside of the Lingua Franca code (like the **physical actions** named `external` in Figure 2b and `ext_open` in Figure 7b) can be modeled as message servers that are invoked nondeterministically. This nondeterministic call can be modeled as a self-call from within the same message server, and there is no need to introduce an extra actor to model the environment. This message server is first called in the constructor of the rebec, as shown for `external` on line 8 of Figure 2c, and for `ext_open` on line 32 of Figure 7c.

Because the Timed Rebeca code will be used for model checking, we need to be careful regarding the state space explosion. The external method calls can be problematic here, and the Timed Rebeca models may have to be carefully crafted in some places. The logical time intervals over which these methods can be called has a great effect on the state space size. If the state space gets too large, model checking becomes intractable.

Although we performed the mapping from Lingua Franca to Timed Rebeca by hand, it should be possible to create a Rebeca target for Lingua Franca and then automate the translation. When using this target, the body of each reaction will need to be written in Rebeca's own language for writing message servers. This is necessary because Afra analyzes this code to build the transition system model, and as for now Afra is not capable of analyzing arbitrary C, C++, or TypeScript code, the target languages currently supported by Lingua Franca.

One subtle point in model checking of CPS that we presented is that we only checked how the state of the program evolves in logical time, not how it evolves in physical time. Every model checking tool that we know of assumes a single timeline, but our systems always have at least three. There is the logical timeline of timestamps, and programs can be verified on this timeline, proving for example that a safety condition is satisfied by a state trajectory evolving on this logical timeline. But in a concurrent and distributed CPS, the state trajectory is also evolving along a physical Newtonian timeline, and our proof says nothing about its safety on that timeline. Moreover, every clock that measures Newtonian time will differ from every other clock that measures Newtonian time, so any constraints we impose on execution based on such clocks may again lead to proofs of safety even though the physical system is capable of entering unsafe states. Our approach in this paper is relying on a set of assumptions mainly based on alignment of logical and physical time at the execution time.

When we assert that a design has been "verified" against a set of formal requirements, we need to make every effort to make as clear as possible what are the assumptions about the physical system that make our conclusions valid. There will *always* be assumptions, and in any real system deployment, *any* assumption may be violated. There is no such thing as a provably correct system.

Funding: The work of the first author is supported in part by KKS SACSys Synergy project (Safe and Secure Adaptive Collaborative Systems), and KKS DPAC Project (Dependable Platforms for Autonomous Systems and Control) at Mälardalen University, and MACMa Project (Modeling and Analyzing Event-based Autonomous Systems) at Software Center, Sweden. The second author receives support from the National Science Foundation (NSF), award #CNS-1836601 (Reconciling Safety with the Internet) and the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by Camozzi Industries, Denso, Siemens, and Toyota.

References

1. Lee, E.A. Cyber Physical Systems: Design Challenges. Int. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC). IEEE, 2008, pp. 363 – 369. doi:10.1109/ISORC.2008.25.
2. Sirjani, M.; Provenzano, L.; Asadollah, S.A.; Moghadam, M.H. From Requirements to Verifiable Executable Models using Rebeca. International Workshop on Automated and verifiable Software sYstem DEvelopment, 2019.
3. Lee, E.A.; Seshia, S.A. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*; MIT Press, 2017.
4. Alur, R. *Principles of Cyber-Physical Systems*; MIT Press, 2019.
5. Henzinger, T.A. The Theory of Hybrid Automata. Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, 1996. IEEE Computer Society, 1996, pp. 278–292.
6. Carloni, L.P.; Passerone, R.; Pinto, A.; Sangiovanni-Vincentelli, A. Languages and Tools for Hybrid Systems Design. *Foundations and Trends in Electronic Design Automation* **2006**, *1*, 1–204. doi:10.1561/1000000001.
7. Sirjani, M.; Movaghar, A.; Shali, A.; de Boer, F.S. Modeling and Verification of Reactive Systems using Rebeca. *Fundam. Inform.* **2004**, *63*, 385–410.
8. Sirjani, M.; Jaghoori, M.M. Ten Years of Analyzing Actors: Rebeca Experience. Formal Modeling: Actors, Open Systems, Biological Systems, 2011, pp. 20–56.
9. Rebeca. Afra Tool, 2019. Available at http://rebeca-lang.org/alltools/Afra, Retrieved July, 2019.
10. Lohstroh, M.; Schoeberl, M.; Goens, A.; Wasicek, A.; Gill, C.; Sirjani, M.; Lee, E.A. Invited: Actors Revisited for Time-Critical Systems. Design Automation Conference (DAC), 2019.
11. Lohstroh, M.; Romeo, I.n.I.; Goens, A.; Derler, P.; Castrillon, J.; Lee, E.A.; Sangiovanni-Vincentelli, A. Reactors: A Deterministic Model for Composable Reactive Systems. Model-Based Design of Cyber Physical Systems (CyPhy'19), 2019. Held in conjunction with ESWEEK 2019.
12. Lohstroh, M.; Lee, E.A. Deterministic Actors. Forum on Specification and Design Languages (FDL),, 2019.
13. Hewitt, C. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence* **1977**, *8*, 323–363.

14. Agha, G.A. *ACTORS - a model of concurrent computation in distributed systems*; MIT Press series in artificial intelligence, MIT Press: Cambridge, MA, 1990.
15. Reynisson, A.H.; Sirjani, M.; Aceto, L.; Cimini, M.; Jafari, A.; Ingólfsdóttir, A.; Sigurdarson, S.H. Modelling and simulation of asynchronous real-time systems using Timed Rebeca. *Sci. Comput. Program.* **2014**, *89*, 41–68.
16. Sirjani, M. Rebeca: Theory, Applications, and Tools. Formal Methods for Components and Objects, International Symposium, FMCO 2006, 2006, pp. 102–126.
17. Derler, P.; Lee, E.A.; Matic, S. Simulation and Implementation of the PTIDES Programming Model. International Symposium on Distributed Simulation and Real-Time Applications, 2008, pp. 330–333.
18. Corbett, J.C.; et.al. Spanner: Google Globally Distributed Database. *ACM Trans. Comput. Syst.* **2013**, *31*, 8:1–8:22.
19. Sirjani, M.; Khamespanah, E.; Lee, E.A. Model Checking Software in Cyberphysical Systems. COMPSAC 2020, 2020.
20. Benveniste, A.; Berry, G. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE* **1991**, *79*, 1270–1282.
21. Edwards, S.A.; Lee, E.A. The Semantics and Execution of a Synchronous Block-Diagram Language. *Science of Computer Programming* **2003**, *48*, 21–42. doi:10.1016/S0167-6423(02)00096-5.
22. Zhao, Y.; Lee, E.A.; Liu, J. A Programming Model for Time-Synchronized Distributed Real-Time Systems. Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2007, pp. 259 – 268. doi:10.1109/RTAS.2007.5.
23. Corbett, J.C.; Dean, J.; Epstein, M.; Fikes, A.; Frost, C.; Furman, J.; Ghemawat, S.; Gubarev, A.; Heiser, C.; Hochschild, P.; Hsieh, W.; Kanthak, S.; Kogan, E.; Li, H.; Lloyd, A.; Melnik, S.; Mwaura, D.; Nagle, D.; Quinlan, S.; Rao, R.; Rolig, L.; Saito, Y.; Szymaniak, M.; Taylor, C.; Wang, R.; Woodford, D. Spanner: Google's Globally-Distributed Database. OSDI, 2012. doi:10.1145/2491245.
24. Lingua Franca Grammar. Lingua Franca Github . Available at https://github.com/icyphy/lingua-franca/blob/master/xtext/org.icyphy.linguafranca/src/org/icyphy/LinguaFranca.xtext, Retrieved May, 2020.
25. Schneider, C.; Spönemann, M.; von Hanxleden, R. Just Model! – Putting Automatic Synthesis of Node-Link-Diagrams into Practice. Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13); , 2013; pp. 75–82. doi:10.1109/VLHCC.2013.6645246.
26. Hardebolle, C.; Boulanger, F. ModHel'X: A Component-Oriented Approach to Multi- Formalism Modeling. MODELS 2007 Workshop on Multi- Paradigm Modeling. Elsevier Science B.V., 2007.
27. Ptolemaeus, C. *System Design, Modeling, and Simulation using Ptolemy II*; Ptolemy.org: Berkeley, CA, 2014.
28. Jantsch, A. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*; Morgan Kaufmann, 2003.
29. Sirjani, M.; Khamespanah, E. On Time Actors. Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday, 2016, pp. 373–392.
30. Khamespanah, E.; Sirjani, M.; Sabahi-Kaviani, Z.; Khosravi, R.; Izadi, M. Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. *Sci. Comput. Program.* **2015**, *98*, 184–204.
31. International Electrotechnical Commission. *International Standard IEC 61131: Programmable Controllers*, 4.0 ed.; IEC, 2017.
32. Berger, H. *Automating with SIMATIC S7-1500: Configuring, Programming and Testing with STEP 7 Professional*, 1st ed.; Publicis MCD Werbeagentur GmbH, 2014.