Model Checking Software in Cyberphysical Systems

Marjan Sirjani
School of IDT
Mälardalen University
Västerås, Sweden
marjan.sirjani@mdh.se

Ehsan Khamespanah
ECE Department
University of Tehran
Tehran, Iran
ekhamespanah@ut.ac.ir

Edward A. Lee
Department of EECS
UC Berkeley
Berkeley, USA
eal@berkeley.edu

***Abstract*—Model checking a software system is about verifying that the state trajectory of every execution of the software satisfies formally specified properties. The set of possible executions is modeled as a transition system. Each "state" in the transition system represents an assignment of values to variables, and a state trajectory (a path through the transition system) is a sequence of such assignments. For cyberphysical systems (CPSs), however, we are more interested in the state of the physical system than the values of the software variables. The value of model checking the software therefore 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 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. Even this finer-grained model, however, may not be sufficiently faithful, and the transition system model needs to be refined further to express not only the properties of the software, but also the properties of the hardware on which it runs. We illustrate these tradeoffs using a coordination language called Lingua Franca that is well-suited to extracting transition system models at these various levels of granularity, and we extend the Timed Rebeca language and its tool Afra to perform this extraction and then to perform model checking.**

***Index Terms*—Cyberphysical systems, Lingua Franca, Model checking, Rebeca, Verification.**

I. Introduction

Formal verification is about assuring properties of models. Whether such properties are also assured in the physical system being modeled depends on the relationship between the model and the physical system. When verifying software, we can often ignore this relationship because 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 very high probability, the physical system that executes the software will also have that property. The microprocessor that runs the software, after all, was designed to do exactly that. When verifying cyberphysical systems (CPS), where software reads sensor data and issues commands to physical actuators, ignoring this relationship is more dangerous. For CPS, verification is ultimately about assuring properties of the physical world not the software abstraction. This means that it is not sufficient to study the software alone. We need to also study its interactions with its environment.

Of course, there is nothing new about formally studying the interactions of software with its environment. In 1977, for example, Pnueli was developing temporal logics for reactive systems [1]. In 1985, Harel and Pnueli singled out reactive systems as being "particularly problematic when it comes to finding satisfactory methods for behavioral description" [2]. They defined reactive systems to be those that are "repeatedly prompted by the outside world, and their role is to continuously respond to external inputs." Cyberphysical systems are reactive in this sense, but also often proactive, prompting the outside world which then reacts to this software-generated stimulus. The closed-loop interactivity between the physical world and the software is what makes these systems so interesting and challenging.

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. To prove properties of such systems, the ultimate goal of verification, requires combining the "semantics" of physics with that of software. We immediately run into difficulties, however, because the semantic worlds of physics and software are radically different and often mutually incompatible. For example, in formal verification of software, it is common to model the software as a transition system that sequentially moves from one state to another, whereas in physics, there is no such sequential behavior and even defining the notion of "state" can be problematic, as we will show.

When we talk about modeling cyberphysical systems, the problem of combining discrete and continuous models inevitably comes to mind. This problem is well addressed by hybrid systems modeling, analysis, and simulation tools [3] and is not the topic of this paper. For modeling the timing of discrete systems, there is also a richness of tools and techniques, including timed automata [4], [5], timed CCS [6], timed Petri nets [7], and Timed Rebeca [8]. These tools assume that the timing of software components and their communication

fabrics are somehow known or can be effectively modeled nondeterministically. We will highlight the difficulties posed by this assumption and explain when weaker assumptions will suffice.

In this paper, we highlight the difficulties that arise when developing appropriate coupled abstractions of the physical system and the software system. We argue that in order to effectively couple models of software with models of the physical world, we will need to enrich the modeling frameworks so that they have more than one timeline. We will illustrate this by analyzing a particular programming language (Lingua Franca [9]) that includes a notion of "logical time" and binds that notion to "physical time" only where the software interacts with the physical world. We will show that programs in this language can be translated into models in a timed modeling language (an extension of Timed Rebeca [8]) that can be model checked to prove properties about the cyber-physical combination. We will use a simple illustrative application that, despite its astonishing simplicity, collides head first with the modeling problems that we highlight.

A. Running Example: Train Door Controller

Consider a train door that needs to be locked before the train starts moving [10]. 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.

To illustrate this point, consider in Figure 1 the sketch of an implementation of a highly simplified version of such train controller software. This implementation is written using Lingua Franca (LF) [11], [9], [12], a coordination language designed for embedded real-time systems. We will fully justify this choice of language later in this paper. 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 [13] (SCADE, Esterel, Lustre, SIGNAL, etc.), Simulink, LabVIEW, ModHel'X [14], Ptolemy II [15], or ForSyDe [16], 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 2. 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

```
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 }
```

Fig. 1. Lingua Franca code for a very simple door controller example with a potential defect.

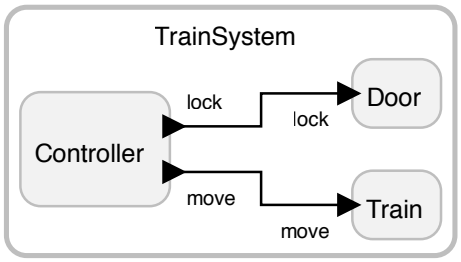

Fig. 2. Structure of the program in Figure 1.

`physical action` named "`external_move`" (line 5), which in Lingua Franca is an event that is triggered by something outside the software system and is then assigned a logical timestamp that approximates the physical time at which that something occurred in the physical world [9]. 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 timestamp. The `reaction` to the `external_move` action (starting on line 9) will be

invoked when logical time reaches the assigned timestamp. 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 timestamp.

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).

B. A Safety Requirement

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. To do so, however, we need a model. And how to construct the "right" model proves astonishingly subtle, even for such a trivial example. Let us examine the problem.

In the program shown in Figure 1, 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. The state of the software system and the state of the physical world are not assured of aligning.

Even if we limit our scope to just the software system, there are traps we could fall into. With such a trivial example, it seems that it should be easy to determine whether the undesired state can ever be reached, but there are subtleties. 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? This question requires us to confront two surprisingly difficult and intertwined topics, time (to resolve the meaning of "simultaneously") and state. What does it mean for a *distributed* system to be in a state?

C. Time and State

We are interested in the dynamics of a system, how it evolves in time. One way to approach this is to adopt the Newtonian view of time, which assumes that time advances uniformly everywhere. We could define the state of a system at a Newtonian time τ to be the state of each of its components at time τ. Let us call such a state a **Newtonian state**.

Modeling a system as the evolution of Newtonian states amounts to a "God's eye view." It assumes an observer that can simultaneously comprehend the state of each of the components at an instant in time. This view is intuitive, but problematic in practice.

In a system involving distributed software, Newtonian state is particularly difficult to use. We could define the state of the software at a time τ to be the values of all the variables in the program at that time. But then, to model the changes of state over time, we need to construct an extremely detailed model of the implementation on which the software is executing. The dynamics (how state evolves in time) is affected by the choice of microprocessor, contention for shared resources such as busses and caches, the scheduling policies of the operating system, and what else is executing on the microprocessors. Moreover, in a distributed system, the temporal alignment between the steps of programs running on distinct microprocessors is not well defined.

The program in Figure 1, despite its simplicity, gives little hint about what that assignment of values to variables at some time τ might be. To have any idea, we need a great deal more information. What microprocessors are running and at what clock rate? How accurate are the clocks? What other network traffic is there? What else is running on the microprocessors? What scheduling policy is being used? Our simple problem has become an astonishingly complex one.

In practice, a model that accurately describes the execution of a distributed program as a function of Newtonian time is intractable. We could abstract the model with nondeterminism, modeling our lack of knowledge about execution in time with probability mass functions. But the result will likely still be intractable because the number of nondeterministic interleavings of state transitions in separate components will explode exponentially, overwhelming any analysis tool. Somehow, we need to reign in the complexity.

To solve this problem, we will have to choose a different definition of "state." Newtonian state will not do for the software components. However, because we are interested in cyber-*physical* systems, not just cyber systems, we cannot completely give up Newtonian time. Newtonian time and Newtonian state are solidly established in the modeling of physical components.

II. Verification Using Model Checking

Model checking is a technique that systematically checks whether a property holds for a model [17]. The model is a transition system, a collection of "states" and transitions between states. There is no requirement that the "states" be Newtonian states. For model checking to be useful, the only requirement is that the transition system faithfully models the system being verified. We can exploit this flexibility to simplify the problem.

In a transition system, the possible behaviors of the system are modeled as traces; each trace begins in some state and sequentially transitions through a sequence of states. In such a model, when there are concurrent components, a common approach is to interleave their transitions. For example, if one component transitions from state A to B concurrently with another component transitioning from C to D, the transition system might model this as a nondeterministic choice between a trace $(A, C) \rightarrow (B, C) \rightarrow (B, D)$ and another trace $(A, C) \rightarrow (A, D) \rightarrow (B, D)$.

Under an interleaving semantics, the semantics of a distributed program is a set of traces representing all allowed interleavings. Every trace in the set is, by definition, a correct execution. There is no requirement that concurrent transitions occur at the same Newtonian time, or even at close Newtonian times.

In an interleaving semantics, concurrent actions are always modeled as sequences of atomic actions. "Atomic" here means that no observer can see a partially executed action. Thus, a transition is logically instantaneous and indivisible. For any concurrent behavior in the physical system, such as software executing on two distinct microprocessors, the model anstracts them as *sequences* of atomic actions.

In a CPS, the actions taken by a piece of software may not really be atomic in this sense. If the software senses or actuates something in the physical world, then an observer in the physical world may in fact witness a partially executed action.

In the theory of concurrent software, one can adopt a different semantic model that replaces interleaved atomic actions with simultaneously evolving behaviors. In the above example, where one component transitions from state A to B concurrently with another component transitioning from C to D, we can define our state transition model in a way that a trace progresses directly from (A, C) to (B, D), without requiring the components to transition in some order. If these transitions take time and have side effects in the physical world, then this model says nothing about the ordering of those side effects.

Model checking is clearly still possible here. The transition system will include the states (A, C) and (B, D) and a transition $(A, C) \rightarrow (B, D)$. There are no states (A, D) and (B, C) in the model. The choice to use this model instead of an interleaving semantics is a choice for the definition of "state" and "transition." To perform model checking, however, we still need a *discrete* transition system model. This means that the transition $(A, C) \rightarrow (B, D)$ must itself be atomic. This again may not be faithful to the physical program execution if the transitions $A \rightarrow B$ and $C \rightarrow D$ are executing on two physically separate microprocessors.

When a model checking tool verifies that a property holds, it provides a proof that the property holds *for the model*, not for the physical realization of the model. We must avoid confusing the map and the territory. It is an error to conclude that the property holds for the physical realization. Any confidence that a model-checking proof might give us must be based on an assessment of how faithful the model is to the physical reality.

Concurrent and distributed programs that do not interact with the physical world can be effectively modeled by interleaving the actions of their single-threaded components, as long as enough care is taken in defining the granularity of the atomic actions. But building a faithful model for concurrent programs that interact with the physical environment is not so straightforward.

Lingua Franca. To help enable such modeling, Lingua Franca includes in its semantics a notion of logical time. Software components are called "reactors." The messages exchanged between reactors have logical timestamps drawn from a discrete, totally ordered model of time. Any two messages with the same timestamp 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 timestamp order. If the reactor has reacted to a message with timestamp t, no future reaction will see any message with a lesser timestamp.

If a reactor produces output messages in reaction to an input, then the logical time of the output will be identical to the logical time of the input. This principle is borrowed from synchronous languages [13]. 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 is a unique least fixed point of a monotonic function on a lattice [18], so the computation is deterministic, even if it is distributed across a network.

Timed Rebeca. Timed Rebeca [8], [19], [20] is an extension of the Reactive Object Language, Rebeca [21], [22], [23], and is designed for modeling and verification of distributed, concurrent and event-driven asynchronous systems with timing constraints. The original Rebeca language models Hewitt actors [24], [25], which do not have a model of time and handle 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.

III. Logical-Time-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**.

With such a semantics, it is easy to verify that the LF program in Figure 1 never reaches the undesired state where the train is moving and the door is unlocked. To perform such verification formally, we need to build a state-transition model

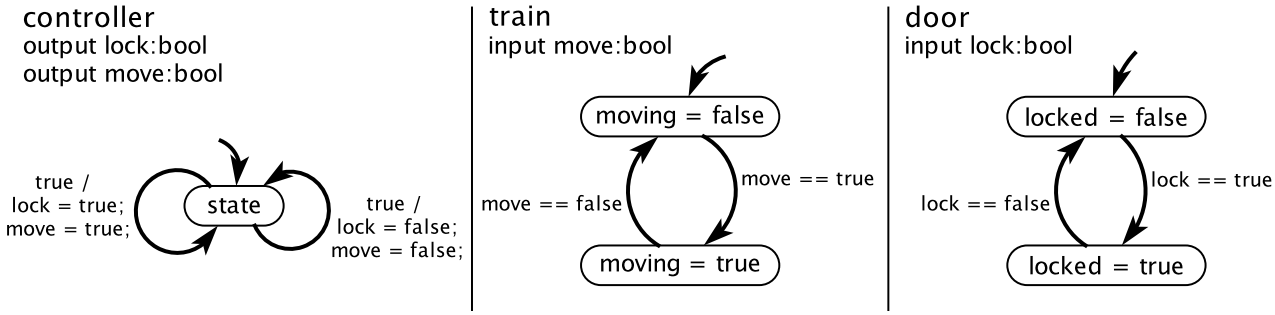

Fig. 3. Concurrent composition of state machine models for each of the reactors in Figure 1.

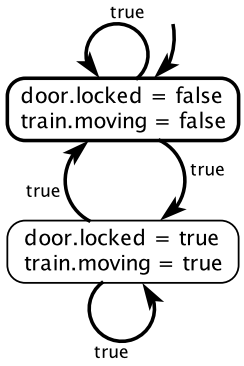

Fig. 4. Semantics of Figure 3 as a single state machine.

of the program. We will first do this manually, and later show how this can be automated.

To build a transition system model, we can use the notation of Lee and Seshia [26] to first model the concurrent composition of reactors as a synchronous composition of state machines, as shown in Figure 3. This notation is based on Statecharts [27], where a vertical bar separates the three concurrent components, the "controller," "train," and "door." These components will be composed according to the principles of synchronous languages.

The left machine has just one state and two transitions. It chooses nondeterministically between these transitions issuing either lock and move both *true* or lock and move both *false*. The middle state machine models the train, which reacts to the move message by moving or stopping. The right machine models the door, which reacts to the lock message by locking or unlocking. According to the principles of synchronous languages, and consistent with the semantics of Lingua Franca, these machines react (logically) simultaneously and instantaneously.

Using the methods of Lee and Seshia [26], we can synchronously combine these state machines to obtain the single state machine shown in Figure 4. This combined state machine gives the semantics of the synchronous composition of the original three machines. In the initial state, indicated by the dangling arrow, the door is unlocked and the train is not moving. At each logical time, this state machine will react by nondeterministically either remaining in the same state (indicated by the self-loop transitions) or changing to the other state (the guards on all transitions are "true," indicating that all are enabled at each logical time). Once the machine 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 [18].

Looking at Figure 4, 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 train component and the door component are realized on distinct microprocessors connected over 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, and any confidence that it gives is misplaced.

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 [28], [29]. 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.

IV. Event-Based Semantics

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 of the previous section in that 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.

For the train door example, an event-based semantics will be more detailed than that of Figure 4. Even for such a small program, it is tedious and error-prone to manually construct such a model. Instead, we have made a small modification to the Timed Rebeca formalism and its tool Afra [30] so that it can model Lingua Franca programs. The modification adds priorities to reactors (called "message servers" in Rebeca) within a reactor (called a "rebec" in Rebeca), and also adds priorities to the actors themselves. This enables capturing all of the scheduling constraints that ensure determinism in a Lingua Franca program.

A (slightly simplified) Timed Rebeca model of the program in Figure 1 is shown in Figure 5. Given this model, we can use Afra model checking tool to get the transition system model and to check safety properties. The event-based transition system is shown in Figure 6.

On line 10, the constructor for the Controller sends itself the message `external`. On line 14 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. If this value is changed from the previous period (comparing `moveP` and `oldMoveP` on line 15) then the two message servers `lock` and `move` are called to lock (or unlock) the door and move (or stop) the train (lines 16 and 17). This `external` message is sent to itself every one time unit by the controller (line 19).

The transitions shown in black in Figure 6 are intermediate transitions that all occur at the same logical time. The transitions shown in red coincide with the advancement of logical time.

```
1  reactiveclass Controller(5) {
2    knownrebecs {
3      Door door;
4      Train train;
5    }
6    statevars {
7      boolean moveP;
8    }
9    Controller() {
10     self.external();
11   }
12   msgsrv external() {
13     boolean oldMoveP = moveP;
14     moveP = ?(true,false);
15     if(moveP != oldMoveP) {
16       door.lock(moveP);
17       train.move(moveP);
18     }
19     self.external() after(1);
20   }
21 }
22 reactiveclass Train(5) {
23   statevars {
24     boolean moving;
25   }
26   Train() {
27     moving = false;
28   }
29   msgsrv move(boolean tmove) {
30     if (tmove) {
31       moving = true;
32     } else {
33       moving = false;
34     }
35   }
36 }
37 reactiveclass Door(5) {
38   statevars {
39     boolean is_locked;
40   }
41   Door() {
42     is_locked = false;
43   }
44   msgsrv lock (boolean lockPar) {
45     is_locked = lockPar;
46   }
47 }
48 main {
49   @priority(1) Controller controller(door,
       train):();
50   @priority(2) Train train():();
51   @priority(2) Door door():();
52 }
```

Fig. 5. Model for the train controller example in Figure 2 constructed in Timed Rebeca extended with priorities.

Thus, Figure 4 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.[1]

In the transition system of Figure 6 [2], 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

[1]The self-loops in Figure 4 are represented as the transitions from S6_0 to S7_0 and back, and S1_0 to S3_0 and back in Figure 6

[2]Figure 6 is generated automatically by Afra and includes some information on the transitions that are irrelevant to our discussion in this paper and the reader may ignore those.

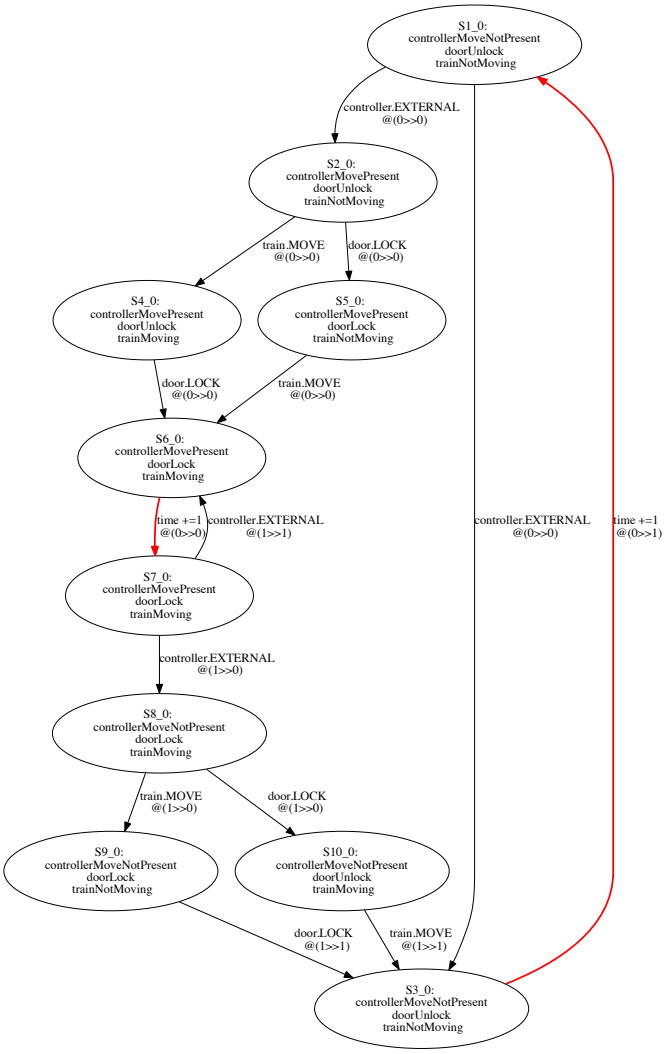

Fig. 6. Transition system model generated from the Timed Rebeca model in Figure 5 generated by Afra [30].

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 4. Without such an I/O system, however, we have more work to do before we can have confidence in this system.

A. *Refining the Program*

The flaw identified by the Afra tool can be corrected with a slightly more sophisticated Lingua Franca program. A simple way to do this is to increment the timestamp of an unlock or move message so that it has a logical timestamp that is strictly larger than the corresponding stop or lock message. Such a Lingua Franca program is shown in Figure 7. It has the structure shown in Figure 8.

```
1  target C;
2  reactor Controller {
3    output lock:bool;
4    output unlock:bool;
5    output move:bool;
6    output stop:bool;
7    physical action external:bool;
8    reaction(startup) {=
9      ... Set up external sensing.
10   =}
11   reaction(external)->lock, move {=
12     if (external_value) {
13       set(lock, true);
14       set(move, true);
15     } else {
16       set(unlock:true);
17       set(stop:true);
18     }
19   =}
20 }
21 reactor Train {
22   input move:bool;
23   input stop:bool;
24   state moving:bool(false);
25   reaction(move) {=
26     self->moving = true;
27   =}
28   reaction(stop) {=
29     self->moving = false;
30   =}
31 }
32 reactor Door {
33   input lock:bool;
34   input unlock:bool;
35   state locked:bool(false);
36   reaction(lock) {=
37     ... Actuate to lock door.
38     self->locked = true;
39   =}
40   reaction(unlock) {=
41     ... Actuate to unlock door.
42     self->locked = false;
43   =}
44 }
45 main reactor System {
46   c = new Controller();
47   d = new Door();
48   t = new Train();
49   c.lock -> d.lock;
50   c.unlock -> d.unlock after 100 msec;
51   c.move -> t.move after 100 msec;
52   c.stop -> t.stop;
53 }
```

Fig. 7. Variant of Figure 1 that manipulates timestamps.

Here, we use the `after` keyword on lines 50 and 51 to increment the timestamp 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 timestamp of the original request and a move message with a timestamp incremented by 100 msec. When it requests that the train stop,

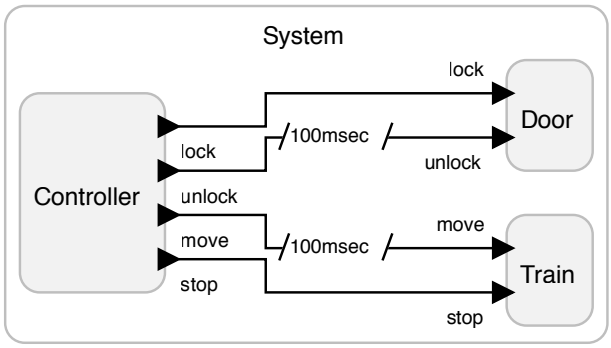

Fig. 8. Structure of the program in Figure 7.

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 2, 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 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 timestamps 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 in a `lock` message to the Door with timestamp 0 and a `move` message to the Train with timestamp 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 timestamp 50 msec, overtaking the `move` message! But worse, it will send an `unlock` message with timestamp 150 msec, and the door will unlock while the train is moving! This new flaw is revealed by a counterexample generated by Afra.

This new flaw is not correctable by simply manipulating logical timestamps. The flaw pertains to the relationship between physical time and logical time (having no constraints on the spacing between timestamps 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 timestamp 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, bur rather by the combination of software and hardware.

No model perfectly represents any physical realization, of course, but there are some key missing elements here that are particular to cyberphysical systems and for which we cannot offer a general solution. We examine those next.

V. Discussions and Limitations

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. 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). We map the reactors of Lingua Franca to rebecs in Timed Rebeca and map reactions to message servers. The connections between inputs and outputs in Lingua Franca show which message servers are called in each rebec. The `after` keywords in both languages increase the timestamp of the messages by the specified value.

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 action** named `external` in Figure 1) 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 on line 10 of Figure 5.

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 Afra is not capable of analyzing arbitrary C, C++, or TypeScript code, the target languages currently supported by Lingua Franca.

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.

A more subtle limitation arises from the fact that we have only checked how the state of the program evolves in logical time, not how it evolves in physical time. We consider it a fascinating open problem to figure out how to adapt today's model checking tools to handle such relations between timelines. 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. We conjecture that model checking tools can be augmented to more effectively handle such a multiplicity of timelines.

A final subtle limitation concerns the need to model a system as a *sequence* of *atomic* actions in order to perform model checking. If the program is executed on a distributed system, then the interleaving semantics implied by a transition system model is faithful to the physical realization only if the atomic actions have no side effects in the physical world or if we go to considerable effort to build a globally synchronous implementation that effectively operates as a single sequential machine.

In the context of Lingua Franca and Rebeca, it is sufficient to assume that every reaction (message handler) is limited to changing the state of its own reactor (actor) and sending output messages. But this will make it difficult to implement a CPS because it prohibits interaction between the software and the physical world. Moreover, if the reaction is implemented in C (or any modern programming language except a pure functional language), then this assumption cannot be enforced by the compiler. Nearly every modern programming language includes, for example, the ability to print messages to a console (`printf` in C, for example). This has a side effect in the physical world (a message appears on the screen). If two independent reactions print a sequence of output strings each, then their execution is not atomic, and an observer may see an arbitrary interleaving of these strings.

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.

REFERENCES

[1] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977. According to Alur and Henzinger, the first application of temporal logic (tense logic) to reactive systems.

[2] David Harel and Amir Pnueli. *On the Development of Reactive Systems*, volume F13 of *NATO ASI Series*, pages 477–498. Springer-Verlag, 1985.

[3] Luca P. Carloni, Roberto Passerone, Alessandro Pinto, and Alberto Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1(1/2):1–204, 2006.

[4] Rajiv Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[5] Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron A. Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer, 1999.

[6] Wang Yi. Ccs + time = an interleaving model for real time systems. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez Artalejo, editors, *Automata, Languages and Programming*, pages 217–228, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[7] Louchka Popova-Zeugmann. *Timed Petri Nets*, pages 139–172. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[8] Arni Hermann Reynisson, Marjan Sirjani, Luca Aceto, Matteo Cimini, Ali Jafari, Anna Ingólfsdóttir, and Steinar Hugi Sigurdarson. Modelling and simulation of asynchronous real-time systems using timed rebeca. *Sci. Comput. Program.*, 89:41–68, 2014.

[9] Marten Lohstroh, Iñigo Incer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Reactors: A deterministic model for composable reactive systems. In *Model-Based Design of Cyber Physical Systems (CyPhy'19)*, Oct. 17-18 2019. Held in conjunction with ESWEEK 2019.

[10] Marjan Sirjani, Luciana Provenzano, Sara Abbaspour Asadollah, and Mahshid Helali Moghadam. From requirements to verifiable executable models using Rebeca. In *International Workshop on Automated and verifiable Software sYstem DEvelopment*, November 2019.

[11] Marten Lohstroh, Martin Schoeberl, Andres Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee. Invited: Actors revisited for time-critical systems. In *Design Automation Conference (DAC)*, June 2019.

[12] Marten Lohstroh and Edward A. Lee. Deterministic actors. In *Forum on Specification and Design Languages (FDL),*, September 2-4 2019.

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

[14] Cécile Hardebolle and Frédéric Boulanger. ModHel'X: A component-oriented approach to multi- formalism modeling. In *MODELS 2007 Workshop on Multi- Paradigm Modeling*. Elsevier Science B.V., October 2 2007.

[15] Claudius Ptolemaeus. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, Berkeley, CA, 2014.

[16] Axel Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann, 2003. Emphasis on modeling, with various MoCs (synchronous, process networks, FSMs, ...) and discussion of heterogeneous mixtures. Describes ForSyDe, a Ptolemy-like framework for heterogeneous MoCs (timed models, synchronous models and untimed models).

[17] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[18] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1):21–42, July 2003.

[19] Marjan Sirjani and Ehsan Khamespanah. On time actors. In *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 373–392, 2016.

[20] Ehsan Khamespanah, Marjan Sirjani, Zeynab Sabahi-Kaviani, Ramtin Khosravi, and Mohammad-Javad Izadi. Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. *Sci. Comput. Program.*, 98:184–204, 2015.

[21] Rebeca. Rebeca Homepage . Available at http://www.rebeca-lang.org/, Retrieved July, 2019.

[22] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundam. Inform.*, 63(4):385–410, 2004.

[23] Marjan Sirjani. Rebeca: Theory, applications, and tools. In *Formal Methods for Components and Objects, International Symposium, FMCO 2006*, pages 102–126, 2006.

[24] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–363, 1977.

[25] Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, Cambridge, MA, 1990.

[26] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. LeeSeshia.org, Berkeley, CA, 2011.

[27] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[28] International Electrotechnical Commission. *International Standard IEC 61131: Programmable Controllers*. IEC, 4.0 edition, 2017.

[29] Hans Berger. *Automating with SIMATIC S7-1500: Configuring, Programming and Testing with STEP 7 Professional*. Publicis MCD Werbeagentur GmbH, 1st edition, 2014.

[30] Rebeca. Afra Tool, 2019. Available at http://rebeca-lang.org/alltools/Afra, Retrieved July, 2019.