To Appear, Design Automation Conference (DAC), June, 2019.

Invited: Actors Revisited for Time-Critical Systems

Marten Lohstroh
UC Berkeley, USA

Martin Schoeberl
TU Denmark, Denmark

Andrés Goens
TU Dresden, Germany

Armin Wasicek
Avast, USA

Christopher Gill
Washington Univ., St. Louis, USA

Marjan Sirjani
Mälardalen Univ., Sweden

Edward A. Lee
UC Berkeley, USA

ABSTRACT

Programming time-critical systems is notoriously difficult. In this paper we propose an actor-oriented programming model with a semantic notion of time and a deterministic coordination semantics based on discrete events to exercise precise control over both the computational and timing aspects of the system behavior.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**;

KEYWORDS

actors, real-time systems, discrete events

ACM Reference Format:
Marten Lohstroh, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee. 2019. Invited: Actors Revisited for Time-Critical Systems. In *Proceedings of The 56th ACM/ESDA/IEEE Design Automation Conference (DAC '19).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Precise timing plays an important role in cyber-physical systems. In order to effectively program these systems, there is a need for models with semantics that includes time. In today's general-purpose hardware and programming languages, timing properties of software are emergent rather than specified. Therefore, the verification of timing properties of time-critical systems relies on overly detailed analysis or testing, but effectively testing software in the face of nondeterminism is challenging and sometimes infeasible.

In this paper, we propose an actor-oriented programming model aimed at time-critical systems. This model reduces nondeterminism by introducing a semantic notion of time and allowing programmers to specify timing properties, which, if executed on capable hardware, can be guaranteed statically. The coordination semantics, based on discrete events, ensures that messages between actors are handled in deterministic order unless nondeterminism is introduced explicitly as a desired property.

DAC '19, June 2–6, 2019, Las Vegas, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 ACTORS

The actor model was introduced by Hewitt [6] in the early 70s. Since then, the use of actors has proliferated in programming languages [1, 2], coordination languages [14, 15], distributed systems [7, 11], and simulation and verification engines [13, 17]. Actors have much in common with objects—a paradigm focused on reducing code replication and increasing modularity via data encapsulation—but unlike objects, actors provide a better model for *concurrency* than threads, the default model for objects. Indeed, each actor is presumed to operate concurrently alongside other actors with which it may exchange messages asynchronously. Objects, in contrast, are often designed assuming a single thread of control, and retrofitting them to be "thread safe" is challenging and error prone. The inherent concurrency of actors makes them ideal for programming reactive systems. However, the lack of any guarantees with respect to the ordering of messages and the absence of a notion of time make this model less useful for specifying systems in which timely execution and repeatable behavior are important.

Extra machinery can be introduced for the formal specification and analysis of systems composed of Hewitt actors. For instance, Real-time Maude [12], a timed rewriting logic framework and temporal model checking tool, has been applied to actors [3]. Similarly, the modeling language Rebeca performs analysis that uses a model checker to ensure that nondeterminism allowed in the model does not lead to behaviors that violate timing requirements [8]. Alternatively, constraints can be placed on actors' allowable behaviors so that they adhere to a well-defined model of computation, satisfying, by construction, desirable properties such as deadlock freedom, schedulability, bounded memory usage, and deterministic execution. It is this latter approach that we assume in this paper.

In [14], Ren and Agha also propose giving actors a temporal semantics. As in our work, they assume a sufficiently well synchronized common physical time base shared by all actors, and they express timing requirements as constraints on message handling. Their work differs from ours, however, in that they build off a standard actor language, thereby inheriting its nondeterministic ordering of message handling, and they rely on separately imposing timing constraints to control the order when needed. In contrast, we use logical timestamps to define the order of message handling and ensure determinism.

Dataflow models are also closely related to the actor model. In [19] the authors extend an (untimed) dataflow model with formal contracts that allow guarantees, e.g., for scheduling. There are timed models of dataflow [18], and even some structured approaches to use timing semantics in dataflow to execute time-critical

```
1  reactor Ramp(p:int(10)) {
2    input set:int;
3    output out:int;
4    clock c(p);
5    constructor {=
6      int count = 0;
7    =}
8    reaction(c) -> out {=
9      count++;
10     set(out, count);
11   =}
12   reaction(set) {=
13     count = set;
14   =}
15 }

16 reactor Print {
17   input in:int;
18   reaction(in) {=
19     printf("%d\n", in);
20   =}
21 }
22
23 composite App {
24   a = new Ramp(p=100);
25   b = new Print();
26   a.out -> b.in;
27 }
```

Figure 1: Example of a `Ramp` feeding into a `Print` reactor

```
1  reactor HTTPGet() {
2    input url:string;
3    output out:string;
4    action arrived:string;
5    reaction(url) {=
6      httpRequest(url, function(reply) {
7        schedule(arrived, reply);
8      });
9    =}
10   reaction(arrived) -> out {=
11     set(out, arrived);
12   =}
13 }
```

Figure 2: Example of a reactor that produces an asynchronous output.

applications in cyber-physical systems [5]. Fredlund et al. proposed timed extension of McErlang as a model checker of timed Erlang programs in [4]. In this extension a new API is introduced to provide the definition and manipulation of timestamps.

3 REACTORS

Our approach in this paper replaces classical actors with a model that we call "reactors," which are similar to actors, but with important differences. We will explain the model first, and then discuss its advantages. We do not make a commitment to a specific syntax for specifying reactors, but, to be concrete, we discuss source code examples written in a language that we are developing and experimenting with.

3.1 Examples

Consider the code in Figure 1. The `Ramp` **`reactor`** emits messages, yielding a counting sequence beginning at 0, at regular time intervals with a period given by its parameter `p`. It declares an **`output`** named `out` of type `int`, a **`clock`** which will trigger reactions periodically, and an **`input`** named `set` of type `int`. The **`constructor`** declares and initializes a state variable named `count`. This reactor has a **`reaction`** triggered by the clock named `c` and one triggered by the input `set`. The first reaction increments the state variable `count` and sends it to the port named `out`, and the second reaction sets the state variable to a specified value. Because there are two reactions, in order to ensure determinacy, the reactor has to define in which order to handle simultaneous triggers (we define "simultaneous" in Section 3.2). Here, we assume that the order of declaration gives that order, so the reaction to `c` will be invoked before the reaction to `set`.

The `Print` reactor has an input port named `in` that triggers its one and only reaction. The **`composite`** named `App` defines one instance of each of these reactors and connects their ports.

The bodies of the reactions are written in some target language; in our example in Figure 1 they are written in C. The target code in the figure is delimited by `{=` and `=}`. A reaction's triggers, as well as any inputs and outputs used in the body of the reaction, are declared. For example, line 8 declares that the reaction is triggered by the clock named `c` and produces an output on the port named `out`.

Reactors can also be used to handle less predictable asynchronous behaviors, such as network interactions, interrupt-driven sporadic sensors, and other unpredictable events. A simple example is sketched in Figure 2. That reactor, upon receiving an input at its `url` port, issues an asynchronous HTTP request over the network. When the reply from the remote server is received, the callback function is invoked. That function schedules a subsequent reaction via the **`action`** named `arrived`. An action is like an input, but messages are not received from other reactors, but rather as a side effect of some action of this reactor itself.

3.2 Principles

We summarize our model with the following principles:

(1) *Components*—Reactors can have input ports, actions, and clocks, all of which are triggers. They can also have output ports, local state, and an ordered list of reactions.
(2) *Composition*—A composite is a reactor that contains other reactors and manages their connections. The connections define the flow of messages, and two reactors can be connected only if they are contained in the same composite. An output port may be connected to multiple input ports, but an input port can only be connected to one output port.
(3) *Events*—Messages sent from one reactor to another, and clock and action events each have a timestamp, a value on a *logical* time line. These are timestamped *events* that can trigger reactions. Each port, clock, and action can have at most one such event at any logical time. An event may carry a value that will be passed as an argument to triggered reactions.
(4) *Reactions*—A reaction is a procedure in a target language that is invoked in response to a trigger event, and *only* in response to a trigger event. A reaction can read input ports, even those that do not trigger it, and can produce outputs, but it must declare all inputs that it may read and output ports to which it may write. All inputs that it reads and outputs that it produces bear the same timestamp as its triggering event. I.e., the reaction itself is logically instantaneous, so any output events it produces are logically simultaneous with the triggering event (the two events bear the same timestamp).
(5) *Flow of Time*—Successive invocations of any single reaction occur at strictly increasing logical times. Any messages that

are not read by a reaction triggered at the timestamp of the message are lost.

(6) *Mutual Exclusion*—The execution of any two reactions of a reactor are mutually exclusive (atomic with respect to one another). Moreover, any two reactions that are invoked at the same logical time are invoked in the order specified by the reactor definition. This avoids race conditions between reactions accessing the reactor state variables.

One of the challenges with understanding our model is that there are two distinct timelines in play, the logical timeline, and an implied physical timeline, or wall-clock time, in which reactions are executed. For example, principle (6) prohibits simultaneous invocation of reactions in physical time, but two reactions triggered at the same stamp are *logically* simultaneous. From principles 3–5 it follows that any reaction of any reactor can only observe input events on different ports if they have the same timestamp. Input ports are *absent* if there is no event with a timestamp matching the time of the reaction. Moreover, any outputs produced by a reaction are logically simultaneous with the triggers. Since, at a logical time, an output can only have one message, if reactions set the output more than once, the last value set will be the one sent at that logical time.

To ensure determinism, each reaction must declare the triggers to which it reacts, any additional input ports that it reads, and to which output ports it may write. These declarations enable a composite to analyze the dependencies between reactions across reactors in order to constrain the order in which reactions are invoked across reactors so that the execution remains deterministic.

Notice that in Figure 2, the reaction to the `url` does not need to declare production of the action `arrived` because that action will be guaranteed (by the `schedule` function) to occur with a strictly greater timestamp than the triggering `url` event.

The `schedule` function in Figure 2 deserves discussion. This is used to activate a trigger at a future logical time. We require that it be a *future* logical time, and consequently it has no impact on the ordering constraints of scheduling of reactions *at* a logical time.

Together, the stated principles ensure that the execution of a composition of reactors is deterministic in the sense that given any set of time-stamped inputs from outside the composition, the composition has exactly one behavior. Within this framework, there are various opportunities for "syntactic sugar," where complex behaviors can be expressed more compactly, for example with the Ptolemy II notion of "multiports" or "persistent ports" [13].

3.3 Discussion

In the original Hewitt actors and many modern implementations, actors have references to the actors that they communicate with. In our approach, rather than addressing each other directly, actors exchange messages via ports. This level of indirection allows actors to be agnostic to the presence or absence of their counterparts. The connections between actors are embedded in a level of hierarchy—a composite—that is responsible for transporting messages between contained actors and constraining the order of invocation of reactions. This approach increases modularity and, more importantly, exposes dependencies between actors. A composite is an entity that, at all times, has a consistent view of the dependencies between the actors it contains, even as the interconnection structure changes and reactors are created and destroyed. The composite mediates all such changes.

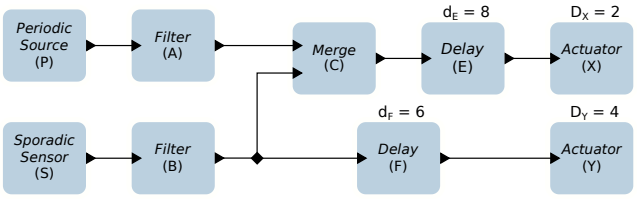

Figure 3: A controller with predefined latencies from reported sensor values to the triggering of actuation reactions

Our model conforms with a well-understood discrete-event (DE) semantics [9]. DE semantics is formally rooted in that of synchronous/reactive models of computation [10].

The functionality of a reaction can be treated as a black box; a schedule can be devised purely based on dependency information, although there is a cost. If a reaction declares that it reads an input, for example, then it will be scheduled at a model time only after that input is known. If it then does not actually read the input, due to a data dependency, then the constraint was unnecessary. We feel that this slight loss in generality is worth the price, particularly because this feature opens the possibility for a *polyglot* language design. A variety of target languages can be supported by the same model. For example, using C++ as a target language is appropriate for the most primitive, resource constrained, deeply embedded systems, while Python may be a better choice for AI applications and Java for enterprise-scale distributed applications running in server farms.

4 TIME, DELAYS, AND DEADLINES

Central to our programming model is the relationship between the timestamps of events. These timestamps denote *logical* time, and the essential semantic feature is that every reactor sees events in timestamp order. When a reactor observes multiple events with the same timestamp, these messages are logically *simultaneous*, and the reactor handles them in a well-defined deterministic order. Logical time is distinct from *physical time*, as measured anywhere in a networked application. First, logical time remains constant during the execution of any reaction in any reactor, and outputs produced by that reaction have the same logical timestamp as the inputs that trigger the reaction. But if we establish a relationship between logical time and physical time, that can help to reliably deliver real-time behavior.

Consider the example in Figure 3, which shows a composition of reactors that serves as a controller in a closed-loop cyber-physical system. This composition has a periodic source P, which could be a sensor taking regularly-spaced samples, for example. It also has a sporadic sensor S that will emit events with some minimum time gap between events. This could be implemented, for example, by an interrupt that causes an invocation of the `schedule` function, like that in Figure 2.

The outputs of both sources have to be timestamped, and it seems like it would be useful in a real-time system if those timestamps reflect the physical time at which sensor measurements were taken.

How can we accomplish that while preserving the determinism of the model? In particular, a timestamp produced by the sporadic sensor will have to be strictly greater than all timestamps seen already by any reactors downstream of it.

We give a simple execution policy that ensures that the principles in Section 3.2 are obeyed. First, logical time and physical time are represented in the same units and aligned at the start of execution of any composition. Second, logical time is never allowed to get ahead of physical time (this constraint can be relaxed, but it must be done carefully). Third, time stamps must be carefully assigned to any new events injected into the system, for example as a consequence of an interrupt.

Suppose an interrupt occurs at some physical time T. Let t denote the current logical time of any executing reaction or the logical time of the most recently executed reaction if no reaction is currently executing. Then a newly created event can be assigned the smallest timestamp that is strictly greater than t and greater than or equal to T. This is the timestamp closest to T that is safe to assign, in the sense that the new event will not appear at any reaction out of timestamp order.

It is possible, of course, for logical time to lag seriously behind physical time. If the composite, for example, simply includes too much computation for the execution to keep up, then this time gap can grow without bound. Such a situation would be disastrous for a real-time system. To prevent it, we introduce the concept of a *deadline* at reactors that have real-time constraints. Typically, these reactors wrap actuators; given input data, they drive physical devices such as motors according to their inputs. Figure 3 has two such actuators, X and Y, with deadlines D_X and D_Y. A deadline D_X on a reaction is interpreted as a constraint that if there is a trigger for that reaction with timestamp t, then the reaction should be invoked no later than physical time $t + D_X$. Given a dependence graph for a composite, sporadic constraints on external sources of events (such as sensors), and worst-case execution time bounds for reactions, it is possible, in principle, to analyze a graph for its ability to meet the deadline. Of course, if the target language is Turing complete, no such analysis can ever be complete. But such analyses are well-understood and form a major part of any assurance effort for a real-time system.

Figure 3 also includes two Delay reactors. These use the `schedule` function to increment the timestamp of their inputs. Given an input with timestamp t, the lower Delay reactor F, for example, will produce an output with timestamp $t + 6$. As logical time is never allowed to get ahead of physical time, a Delay reactor may align logical time with physical time. Hence, if the sporadic sensor S produces an event with timestamp t, the actuator Y is required to process any consequent event no earlier than physical time $t + 6$ and no later than physical time $t + 10$. At the cost of requiring more detailed analysis and possibly specialized microprocessors, such as PRET [21] or Patmos [16], this time window can be made as small as necessary to meet even the most demanding requirements for safety-critical real-time systems.

Coordinating execution of a composite across a distributed platform will require some care. The model we have given here, however, is fully compatible with PTIDES [20], which gives a practical and realizable distributed execution mechanism.

ACKNOWLEDGMENTS

The work in this paper was supported in part by the National Science Foundation (NSF), award #CNS-1836601 (Reconciling Safety with the Internet) and the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by Avast, Camozzi Industries, Denso, Ford, Siemens, and Toyota This work was also supported in part by the Center for Advancing Electronics Dresden (cfaed) and the German Academic Exchange Service (DAAD).

REFERENCES

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

[2] Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., and Zufferey, D. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices 48*, 6 (2013), 321–332.

[3] Ding, H., Zheng, C., Agha, G., and Sha, L. Automated verification of the dependability of object-oriented real-time systems. In *2003 The Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems* (Oct 2003), pp. 171–171.

[4] Earle, C. B., and Fredlund, L. Verification of timed erlang programs using mcerlang. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings* (2012), pp. 251–267.

[5] Geilen, M., Stuijk, S., and Basten, T. Predictable dynamic embedded data processing. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* (2012), IEEE.

[6] Hewitt, C., Bishop, P. B., and Steiger, R. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973* (1973), pp. 235–245.

[7] Hunt, J. *Further Akka Actors.* Springer International Publishing, Cham, 2018, pp. 345–360.

[8] Khamespanah, E., Sirjani, M., Kaviani, Z. S., Khosravi, R., and Izadi, M.-J. Timed rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. *Science of Computer Programming 98* (2015), 184 – 204.

[9] Lee, E. A., Liu, J., Muliadi, L., and Zheng, H. Discrete-event models. In *System Design, Modeling, and Simulation using Ptolemy II*, C. Ptolemaeus, Ed. Ptolemy.org, 2014.

[10] Lee, E. A., and Zheng, H. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT* (2007), ACM, pp. 114 – 123.

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

[12] Ölveczky, P. C., and Meseguer, J. The real-time maude tool. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 332–336.

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

[14] Ren, S., and Agha, G. A. RTsynchronizer: language support for real-time specifications in distributed systems. *ACM Sigplan Notices 30*, 11 (1995), 50–59.

[15] Ren, S., Yu, Y., Chen, N., Marth, K., Poirot, P.-E., and Shen, L. Actors, roles and coordinators — a coordination model for open distributed and embedded systems. In *Coordination Models and Languages* (Berlin, Heidelberg, 2006), P. Ciancarini and H. Wiklicky, Eds., Springer Berlin Heidelberg, pp. 247–265.

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

[17] Sirjani, M., Movaghar, A., Shali, A., and de Boer, F. S. Modeling and verification of reactive systems using rebeca. *Fundam. Inform. 63*, 4 (2004), 385–410.

[18] Sriram, S., and Bhattacharyya, S. S. *Embedded Multiprocessors: Scheduling and Synchronization.* Marcel Dekker, Inc. (now Taylor and Francis), 2000.

[19] Wiik, J., Ersfolk, J., and Waldén, M. A contract-based approach to scheduling and verification of dynamic dataflow networks. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)* (2018), IEEE, pp. 1–10.

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

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