Actors, Time Stamps, and Determinism for Time-Critical Systems

Marten Lohstroh
marten@eecs.berkeley.edu
UC Berkeley, USA

Martin Schoeberl
masca@dtu.dk
TU Denmark, Denmark

Andrés Goens
andres.goens@tu-dresden.de
TU Dresden, Germany

Armin Wasicek
armin.wasicek@avast.com
Avast, USA

Christopher Gill
cdgill@wustl.edu
Washington Univ., St. Louis, USA

Marjan Sirjani
marjan.sirjani@mdh.se
Mälardalen Univ., Sweden

Edward A. Lee
eal@eecs.berkeley.edu
UC Berkeley, USA

ABSTRACT

Abstract

CCS CONCEPTS

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

KEYWORDS

actor, real-time systems, worst-case execution time

ACM Reference Format:
Marten Lohstroh, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee. 2019. Actors, Time Stamps, and Determinism for Time-Critical Systems. In *Proceedings of The 56th ACM/ESDA/IEEE Design Automation Conference (DAC '19).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Precision timing plays an important role in a plethora of modern systems, ranging anywhere from embedded control systems that continually interact with concurrent physical processes (i.e., cyber-physical systems) to large-scale distributed systems requiring some measure of consistency. In order to effectively program these systems, there is a need for programming models with a semantics that includes time. In current-day 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 testing, but effectively testing software in the face of non-determinism is challenging.

In this paper, we propose an actor-oriented programming model that reduces nondeterminism by introducing a semantic notion of time, allowing programmers to specify timing properties, which, if executed on capable hardware, can be guaranteed statically. Aimed at time-critical systems, our model is based on a discrete event semantics, which ensures that messages between actors are handled in deterministic order unless nondeterminism is introduced explicitly as a desired property. We introduce *Lingua Franca* (LF), an interface definition language for the description of actors and their composition. LF allows programmers to implement the functionality of actors using a language of their choice. By default, all Lingua Franca programs will be deterministic. We achieve this using a semantic notion of time and time stamps.

2 ACTORS

The actor model was introduced by Hewitt [11] in the early 70s. Since then, the use of actors has proliferated in programming languages [2, 4, 10], coordination languages [1, 22], distributed systems [12, 18], and simulation engines [21, 25]. Marjan ►*Reo is not actor-based.*◄ Marjan ►*BTW, do you want to say actor-oriented or actor-based?*◄ Actors have much in common with objects—a paradigm focused on reducing code replication by means of inheritance and increasing modularity via data encapsulation—but unlike objects, actors provide a better model for *concurrency* than threads, the default model for objects. Marjan ►*We do have concurrent objects, and as far as I remember they are as old as objects and actors.*◄ Edward ►*I changed this to "a better model ...*◄ Marjan ►*Great!*◄ Indeed, each actor is presumed to operate concurrently alongside other actors with which it may exchange messages asynchronously. Objects, in contrast, are usually designed assuming a single thread of control, and retrofitting them to be "thread safe" is challenging and error prone. These properties make actors 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 [20], a timed rewriting logic framework and (temporal) model checking tool, has been applied to actors in [5]. 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 requirements [24]. Marjan ►*, and for the timed version in [13].*◄ Alternatively, constraints can be placed on actors' allowable behaviors so that they adhere to a well-defined model of computation (MoC), satisfying desirable properties such as deadlock freedom, schedulability, bounded memory usage, and deterministic execution, by construction. It is the latter approach that we follow.

3 LINGUA FRANCA

In the original Hewitt actors and many modern implementations, actors have references to the actors that they communicate with. In Lingua Franca, 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. While this approach increases modularity and, more importantly, exposes dependencies between actors. These dependencies can be used to devise a schedule that ensures that (1) actors observe produced messages in time stamp order, and (2) actors cannot produce messages with a time stamp t until all anti-dependent inputs with time stamp t are known. This is the approach taken in the Discrete Event implementation of Ptolemy II [14], which is formally based on a synchronous/reactive MoC [16]. To increase efficiency, our approach avoids performing a fixed-point computation by executing actors in topological order. That way, each actor is only executed at most once per tick of the logical clock. Because the schedule is based purely on the topology of the actor graph, no static analysis of the actors' internals is required—Ptolemy actors are treated as black boxes. Dynamic changes in the topology are handled hierarchically, so that the extent of the graph that is subject to rescheduling is contained when mutations occur during runtime. A runtime mutation always occurs within a composite actor whose interface remains unchanged.

Edward ►*This paragraph needs work.*◄ LF actors, on the other hand, are better characterized as *gray* boxes considering they reveal some key features of their internal structure, namely a closer approximation of the true dependencies between their ports. Edward ►*This is not quite right. The causality interfaces of Ptolemy II (Zhou et al.) expose fine-grain causality information.*◄ Other than actors in Ptolemy II, which implement an abstract semantics [27] that requires each actor to perform all of its computation in a so-called *fire* function, LF actors are broken down into smaller units we call *reactions*. This approach, inspired by the characterization of dataflow processes given in [15], exposes the dependencies between actors' inputs and output at a finer grain. We enforce simple lexical scoping rules, which virtually any programmer is already familiar with, to limit reactions' access to the actors' input and output ports, thereby eliminating dependencies between ports that are out of scope.

The interface definitions of reactions are much like function definitions. Consider the following example which defines a brake component as an LF-actor: Edward ►*A brake component is probably not a good example for an ECMA 6/Flow target language. The target language should be C, or a web example should be chosen. Also, this example won't compile. What is "pressed"? Also, "state" needs to be set in initialize. Also, our current syntax terminates lines with semicolons.*◄

```
language ECMA6/Flow;
actor Brake {%
  input check:any
  input apply:number
  output on:boolean
  preamble {%
    state = false
    actuate() { ... }
  %}
  reaction(apply) -> {%
    pressed.set(true)
    if (apply > 5)
      actuate(apply)
    else
      state = false
  %}
  reaction(check) -> on {%
    on = state
  %}
%}
```

TODO ►*Explain this, discuss polyglot*◄ TODO ►*Precisely because no static analysis is required...*◄

Having this finer-grained dependency information available at the interface level also opens up avenues for scheduling optimizations, makes the detection of zero-delay feedback less conservative, and, we conjecture, will enable remote procedure calls (RPCs) between actors without compromising determinism or modularity. The same dependency analysis performed to devise a schedule for message delivery and rule out zero-delay feedback can also be used to coordinate RPCs and guarantee deadlock freedom. We think that integrating RPCs into LF will be key to achieving wide adoption of our programming model, because it is such an omnipresent language feature, even in some actor-based frameworks, for example Ray [18]. Marjan ►*When you have atomic execution of reactions, then what does RPC mean? what is the difference between an RPC and Call-Backs (or maybe callbacks with priorities)? Maybe we just leave this out for this version of the paper?*◄ Edward ►*I agree that this does not say enough to be understandable. An example would be needed.*◄

Central to the LF programming model is the relationship between the time stamps of messages. These time stamps denote *logical* time, and the essential semantic feature is that every actor sees messages in time-stamp order. When an actor receives multiple messages with the same time stamp, these messages are logically *simultaneous*, and the actor 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 actor, and outputs produced by that reaction have the same logical time stamp as the inputs that trigger the reaction. But in LF, there is a relationship between logical time and physical time, a relationship that helps to reliably deliver real-time behavior. Edward ►*This relationship needs to be explained more carefully and using examples. The following is just going to be confusing. I suggest deleting it and explaining later.*◄ The passing of *physical* time, approximated by the system time of the execution platform. While logical time can be sped up or slowed down arbitrarily in a simulation, the fact that actors can interact with the physical world through sensors and actuators requires that logical time and physical time "keep up" with one another in order to have a temporal semantics that obeys the principle of causality (i.e., effects cannot precede causes) for all observers.

We summarize our model in terms of the following principles:

(1) Messages exchanged between actors are timestamped and denote a discrete event.
(2) Actors have input ports, triggers, output ports, state, and an ordered list of reactions.

(3) A reaction is a procedure, and any two reactions of an actor are mutually exclusive; they execute atomically with respect to one another;
(4) Each reaction declares the input ports and triggers to which it reacts and to which output ports it may write.
(5) If a reaction declares that it reacts to multiple input ports, then at the time of the reaction, at least one of those input ports will have a message time-stamped with the time of the reaction, but the other input ports may be *absent* (no event is present at this logical time).
(6) A reaction must also declare inputs that it does not react to but that it reads; at a logical time, those inputs may also be absent.
(7) At each logical time for which there is an input message or trigger, all reactions to those inputs and triggers will be invoked in the order in which the reactions are defined.
(8) Logical time is constant during execution of a reaction.
(9) For each reaction, logical time is strictly increasing; that is, a reaction will never be invoked at a logical time that is equal to or less than that of a previous invocation.
(10) A reaction can schedule a trigger at a future logical time.
(11) A reaction can read and modify the actor's state.
(12) A reaction can set the values of the output ports that is it has declared, and if it does so, a message will be sent after the conclusion of the reaction with a payload equal to the final value set by the reaction.
(13) If multiple reactions write to the same output port, then the message sent will be that set by the last reaction invoked.
(14) No two distinct actors may send messages to the same input port of another actor; a composition of actors that does this is an erroneous composition.
(15) An output port may send messages to multiple input ports or multiple actors; these messages all bear the same time stamp.

Together, these rules ensure that the execution of a composition of actors is deterministic in the sense that given any set of time-stamped inputs from outside the composition, the composition has exactly one behavior. This determinism can be proved by showing that at each logical time, every actor realizes a monotonic function in an appropriately chosen partial order and that the execution converges to a least fixed point in this partial order [16]. Within these rules, 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" [21].

4 TIMING AND DETERMINISM

TODO ►*discuss timing and scheduling using actor graph examples (no code)*◄

4.1 Delays and Deadlines

𝒟 We have a notion of delay in two places: (1) as a delay actor to break up feedback loops and (2) at actuators to specify that an actuation shall happen before (or exactly at) the logical time of the input plus the delay.

Delays deserve also the purpose to align events with wall clock time and to allow execution of reactions in their worst-case execution time (WCET). The sum of all reaction's WCET from one synchronization point (either sample of sensors or output of a delay actor) to another synchronization point (or output of an actuator) has to be less than the delay.

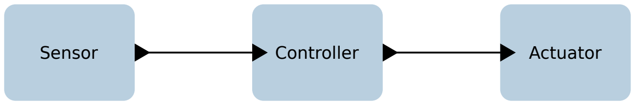

Figure 1: First example

4.2 Sporadic Events and Call Backs

Sporadic events and call backs can trigger a reaction. This reaction is allowed to observe and also change the state of the actor. Therefore, they are not allowed to preempt a reaction (reactions are atomic to each other). To enforce this atomicity a sporadic reaction is executed only between two logical timestamps t_1 and t_2. This can be enforced on a C based host by turning off interrupts when executing a reaction. Interrupts are enabled when all[1] reactions for timestamp t_1 have been executed and disabled again when logical time is advanced to t_2 where all reactions for t_2 are executed.

4.3 Causality

We do not want to run ahead of realtime, because actors can produce spontaneous events that need to be stamped with pysical time, and we don't want these timestamps to be "in the past."

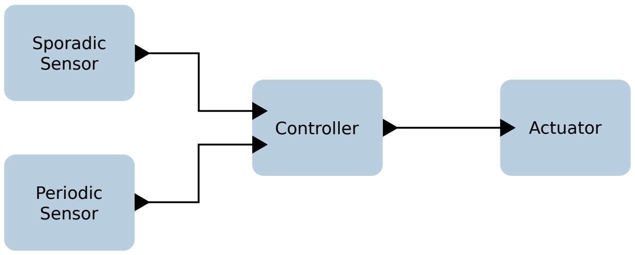

Figure 2: Second example

4.4 Example 3

Shut off the lights some time after the switch has been flipped. Reason to have the deadline definition as stated: detectability. Suppose the start deadline cannot be met; the reaction should not be carried out (and the violation be reported on subsequently).

Marten ►*In third example, discuss how persistent ports can be used.*◄
Marten ►*Also, discuss optimized scheduler that runs ahead of physical time.*◄

[1] TODO ►*All is a little bit restrictive, just for state atomicity we would not need to wait for all reaction, just for a single. But there is another issue that this shall happen only between two timestamps. I don't recall the details.*◄

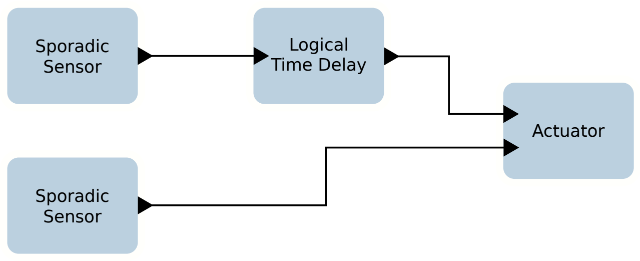

Figure 3: Third example

5 RELATED WORK

Towards a real-time coordination model for mobile computing [9] Method and tools for mixed-criticality real-time applications within PharOS [17] Accessors [3] S-Net [8] Guarded Atomic Actions [23] Timed C [19]

Dataflow models are also closely related to the actor approach. In [28] the authors extend an (untimed) dataflow model with formal contracts that allow guarantees, e.g. for scheduling. There are timed models of dataflow [26], and even some structured approaches to use timing semantics in dataflow to execute time-critical applications in cyber-physical systems [7]. These models are more restrictive than our propsed work and lack the polyglot flexibility of LF.

Fredlund et al. proposed timed extension of McErlang as a model checker of timed Erlang programs in [6]. In this extension a new API is introduced to provide the definition and manipulation of time-stamps. In [?], de Boar et al. provided a survey on active object languages but there is no focus on timed features.

Martin ▶*Oh boy, we have way too many references for a 4 pages position paper.*◀ Marjan ▶*That's why I happily didn't add much to the Related Work section. I just wrote the McErlang part, and now I added a line on a survey that can just be commented out for this paper. I think the work of Gul, RTSynchronizer, is very close, although it is very old. Also the Theatre framework is close but can be ignored for now.* ◀

6 THE LINGUA FRANCA COMPILER TOOL CHAIN

TODO ▶*Describe that the single threaded execution of JavaScript enforces the restriction of the call back not interrupting the reaction.*◀

For dynamic systems, e.g., an IoT that keeps running, but in different contexts (e.g., places), we support dynamic reconfiguration of the network of actors. This may be well supported when targeting a dynamic language such as JavaScript, but harder to implement in C.

To support reconfiguration we need a more complete language than the configuration language LF. Therefore, our first iteration of dynamic reconfiguration is done in the host language JavaScript. In that case, the host language needs access to actors and ports. However, we can use scopes to having access to those parts of the framework only in well defined places.

Martin ▶*Will we talk about execution hosts such as PRET machines or Patmos when targeting C and WCET analysis?*◀

7 CONCLUSIONS

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

REFERENCES

[1] Farhad Arbab. 2004. Reo: A Channel-Based Coordination Model for Component Composition. *Mathematical Structures in Computer Science* 14, 3 (2004), 329–366.

[2] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent programming in Erlang* (second ed.). Prentice Hall.

[3] C. Brooks, C. Jerad, H. Kim, E. A. Lee, M. Lohstroh, V. Nouvellet, B. Osyk, and M. Weber. 2018. A Component Architecture for the Internet of Things. *Proc. IEEE* 106, 9 (September 2018), 1527–1542. DOI: http://dx.doi.org/10.1109/JPROC.2018.2812598

[4] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices* 48, 6 (2013), 321–332.

[5] Hui Ding, Can Zheng, Gul Agha, and Lui Sha. 2003. 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*. 171–171. DOI: http://dx.doi.org/10.1109/WORDS.2003.1267505

[6] Clara Benac Earle and Lars-Åke Fredlund. 2012. 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*. 251–267.

[7] Marc Geilen, Sander Stuijk, and Twan Basten. 2012. Predictable Dynamic Embedded Data Processing. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE.

[8] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. 2008. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters* 18, 02 (2008), 221–237.

[9] Gregory Hackmann, Christopher Gill, and Gruia-Catalin Roman. 2005. Towards a real-time coordination model for mobile computing. In *Monterey Workshop*. Springer, 184–202.

[10] Philipp Haller and Martin Odersky. 2009. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* 410, 2-3 (2009), 202–220.

[11] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. 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*. 235–245.

[12] John Hunt. 2018. *Further Akka Actors*. Springer International Publishing, Cham, 345–360. DOI: http://dx.doi.org/10.1007/978-3-319-75771-1_32

[13] Ehsan Khamespanah, Marjan Sirjani, Zeynab Sabahi Kaviani, Ramtin Khosravi, and Mohammad-Javad Izadi. 2015. Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. *Science of Computer Programming* 98 (2015), 184 – 204.

[14] Edward A. Lee, Jie Liu, Lukito Muliadi, and Haiyang Zheng. 2014. Discrete-Event Models. In *System Design, Modeling, and Simulation using Ptolemy II*, Claudius Ptolemaeus (Ed.). Ptolemy.org.

[15] Edward A. Lee and Eleftherios Matsikoudis. 2009. *The Semantics of Dataflow with Firing*. Cambridge University Press. http://ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/ Draft version available at http://ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/.

[16] Edward A. Lee and Haiyang Zheng. 2007. Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems. In *EMSOFT*. ACM, 114 – 123. DOI: http://dx.doi.org/10.1145/1289927.1289949

[17] Matthieu Lemerre, Emmanuel Ohayon, Damien Chabrol, Mathieu Jan, and Marie-Benedicte Jacques. 2011. Method and tools for mixed-criticality real-time applications within PharOS. In *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. IEEE, 41–48.

[18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *CoRR* abs/1712.05889 (2017). arXiv:1712.05889 http://arxiv.org/abs/1712.05889

[19] S. Natarajan and D. Broman. 2018. Timed C: An Extension to the C Programming Language for Real-Time Systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 227–239. DOI: http:

//dx.doi.org/10.1109/RTAS.2018.00031

[20] Peter Csaba Ölveczky and José Meseguer. 2008. The real-time Maude tool. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 332–336.

[21] Claudius Ptolemaeus. 2014. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, Berkeley, CA. http://ptolemy.org/books/Systems

[22] Shangping Ren, Yue Yu, Nianen Chen, Kevin Marth, Pierre-Etienne Poirot, and Limin Shen. 2006. Actors, Roles and Coordinators — A Coordination Model for Open Distributed and Embedded Systems. In *Coordination Models and Languages*, Paolo Ciancarini and Herbert Wiklicky (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 247–265.

[23] Daniel L. Rosenband and Arvind. 2004. Modular Scheduling of Guarded Atomic Actions. In *Proceedings of the 41st Annual Design Automation Conference (DAC ’04)*. ACM, New York, NY, USA, 55–60. DOI: http://dx.doi.org/10.1145/996566.996583

[24] Marjan Sirjani and Mohammad Mahdi Jaghoori. 2011. Ten Years of Analyzing Actors: Rebeca Experience. In *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*. 20–56.

[25] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. 2004. Modeling and Verification of Reactive Systems using Rebeca. *Fundam. Inform.* 63, 4 (2004), 385–410.

[26] S. Sriram and S. S. Bhattacharyya. 2000. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc. (now Taylor and Francis). This book presents a unified and comprehensive theory for analysis and implementation of multiprocessor digital signal processing (DSP) systems. The novelty and utility of this theory center around its careful integration of scheduling, interprocessor communication, and synchronization. This integration is enabled by a new graph-theoretic framework called the "synchronization graph" for analyzing and optimizing multiprocessor configurations for DSP applications.

[27] Stavros Tripakis, Christos Stergiou, Chris Shaver, and Edward A. Lee. 2012. A Modular Formal Semantics for Ptolemy. *Mathematical Structures in Computer Science Journal* to appear (2012). http://chess.eecs.berkeley.edu/pubs/877.html

[28] Jonatan Wiik, Johan Ersfolk, and Marina Waldén. 2018. 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)*. IEEE, 1–10.