Work-in-Progress: Programs with Ironclad Timing Guarantees

Marten Lohstroh
UC Berkeley, USA

Martin Schoeberl
TU Denmark, Denmark

Mathieu Jan
CEA, List, France

Edward Wang
UC Berkeley, USA

Edward A. Lee
UC Berkeley, USA

ABSTRACT

We discuss ongoing work towards a meta-language, execution model, and compiler tool chain that promotes determinism and grants first-class citizenship to the timing aspects of computation.

CCS CONCEPTS

• **Computer systems organization** → **Real-time languages**; **Real-time system specification**; **Embedded systems**;

KEYWORDS

real-time systems, discrete events, scheduling, concurrency, determinism, polyglot, meta-language, compiler, runtime environment

ACM Reference Format:
Marten Lohstroh, Martin Schoeberl, Mathieu Jan, Edward Wang, and Edward A. Lee. 2019. Work-in-Progress: Programs with Ironclad Timing Guarantees. In *2019 International Conference on Embedded Software Companion (EMSOFT'19 Companion), October 13–18, 2019, New York, NY, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3349568.3351553

1 INTRODUCTION

Precision timing plays an important role in a plethora of modern systems, ranging from embedded control systems and robotics to large-scale distributed systems that require 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. According to a recent study by the Barr Group, over 95 percent of embedded-system code today is written in C or C++ [5]. For these systems, the verification of timing properties relies mostly on testing. However, effectively testing software in the face of nondeterminism is extremely challenging.

We have recently proposed a programming model that adopts a logical notion of time and has a concurrent execution semantics that ensures determinism unless nondeterminism is allowed explicitly [**?**], for instance, to handle sporadic events. This model is based on a variant of actors we call *reactors*, which are components that consist of *reactions* that observe and emit timestamped events. Opportunities for parallel execution of reactions are exposed automatically in a precedence graph that is created at compile time.

EMSOFT'19 Companion, October 13–18, 2019, New York, NY, USA

ACM ISBN 978-1-4503-6924-4/19/10.
https://doi.org/10.1145/3349568.3351553

Timing constraints relating to physical time can be specified in the program, which, if executed on capable hardware, can be guaranteed statically. This aspect makes reactors particularly suited for specifying real-time requirements.

In order to realize reactor programs with ironclad timing guarantees, reactions must be amenable to worst-case execution time (WCET) analysis [4], which informs schedulability analysis. Platforms that are optimized for predictable timing allow for tighter bounds on WCET and more accurate release times, allowing for better utilization and tighter synchronization to physical time, respectively. We are currently developing a compiler tool chain for reactors that targets both POSIX-based systems as well as bare-metal time-predictable processors. For the latter it should hold that if a program compiles, it is guaranteed to obey any timing constraints expressed in terms of delays and deadlines.

2 REACTORS

Reactors are software components that borrow concepts from actors, dataflow models, synchronous-reactive models, discrete event systems, object-oriented programming, and reactive programming.

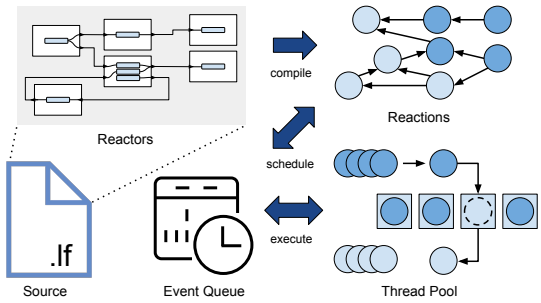

Figure 1: Compilation and execution of reactors

Principles. A reactor has input and output ports, (internal) actions, state, and reactions. Reactors can be composed bilaterally or hierarchically by interconnecting their ports. Unlike classical actors, the communication between reactors is constrained by their connections; their communication topology is represented explicitly, as in dataflow models. Unlike dataflow models, reactors do not have input buffers. Inputs are uniquely defined for each timestamp, as is the case for *ticks* in synchronous-reactive models. As a consequence, outputs produced by reactions have the same timestamp as the inputs that triggered them.

Like actors, reactors do not share state, but reactions within the same reactor do. This approach mirrors the idea of state encapsulation in objects, but, unlike methods, reactions are anonymous and

cannot be invoked directly by other reactions (not even if they are part of the same reactor). Instead, a reaction is sensitive to a set of inputs and/or internal actions, following the *observer* pattern used in reactive programming. Reactions, however, can trigger each other (or themselves) by scheduling an action with some specified delay. If no scheduling delay is specified, the action will occur a microstep later in superdense time. This nuance is important because it makes causality loops detectable.

To rule out non-determinism 1) no two reactions of the same reactor are allowed to execute concurrently, and 2) when two or more reactions of *the same reactor* are triggered at the same logical time, they must execute in a predefined order. Reactions declare dependencies on input ports and actions, and anti-dependencies on output ports. Along with their ordering with respect to other reactions, these declarations provide all the information required to build a precedence graph. This graph encodes the constraints that must be satisfied in order to obey the above two rules, and thus ensure determinacy.

Lingua Franca. Because dependency declarations are part of the interface of reactions, the body of reactions can be treated as a black box. Exploiting this fact, we started developing a polyglot compiler tool chain. While reactors are defined and composed in our own meta-language called *Lingua Franca* (LF), the body of reactions is specified in target language code. The role of the compiler is to generate target code that brings declared ports and actions into reaction scope, and to construct a precedence graph that governs the execution of reactions at any given time step. Cyclic dependencies are detected and reported as error conditions because they represent a causality loop. We are currently developing C and TypeScript back ends for our LF compiler.

Execution Model. As illustrated in Fig. 1, scheduled events enter into an event queue, which is a priority queue that sorts events by timestamp. Execution entails popping events from the event queue and executing all the reactions that it enables; an event can trigger one or more reactions, and each reaction can produce outputs that trigger more reactions. Reactions can also add new events to the event queue. The precedence graph arranges all reactions enabled at a given time into a partial order that informs as to whether a given reaction can be executed or has to wait for any anti-dependent reaction(s) to complete. Since reactions are *partially* ordered, reactions in parallel precedence chains can be executed in parallel.

One of the challenges with understanding our model is that there are two distinct timelines in play: the logical timeline along which events are ordered, and an implied physical timeline, or wall-clock time, in which reactions are executed. In our model, *delays* are associated with actions (i.e., scheduled reactions), and *deadlines* are associated with reactions to input (i.e., synchronous reactions). Actions may be scheduled asynchronously (e.g., in response to the arrival of a sporadic sensor reading or interrupt), and when this happens, the logical time at which the reaction occurs will be derived from the current physical time obtained from the platform. While the execution of each reaction takes physical time, logical time does not advance during reactions. In principle, no event can be processed before physical time exceeds logical time of its triggering event. This is to prevent asynchronously scheduled actions from acquiring a timestamp that is smaller than the current logical time. The extent to which logical time may lag behind physical time at the release of the reaction(s) triggered by some input is specified by a deadline. If accurate execution time bounds can be obtained, then deadlines at actuators imply precise, inferred deadlines upstream in the dependence graph. This allows for the specification of closed-loop deterministic cyber-physical models with predefined latencies from physical sensing to physical actuation. When a deadline is specified, a reaction to a violation of the deadline can also be specified. This enables designs that can react to faults.

Unlike many models involving precedence graphs, not all reactions in the graph are always expected to execute at each logical time. This renders most classical scheduling techniques useless, and opens up an interesting research area for new scheduling algorithms and run time optimizations.

Target Platforms. We have focused our initial efforts on generating C code for reactors that is executable on POSIX-compliant systems. Our current implementation is single-threaded, and preliminary results show that we can handle on the order of 2.5 million reactions per second on a 2.6 GHz Intel Core i7. Our next step is to implement a thread pool and exploit concurrency in the scheduler as depicted in Fig. 1 and evaluate performance.

Our second target is Patmos [3], an architecture that is specifically designed to simplify WCET analysis and is supported by several WCET analysis tools. At this time we have already successfully run reactors on Patmos and have computed WCET for reactions. A closer integration between the LF and Patmos compiler is planned. In particular, the goal of WCET analysis would be to prove that a specified deadline can never be violated.

Lastly, we plan to target Precision-Timed Machines (PRET machines) [**?**] which achieve repeatable timing by using a thread-interleaved pipeline, scratchpad memory instead of caches, and a specialized DRAM controller. FlexPRET [6] distinguishes between soft and hard real-time threads, and supports an arbitrary interleaving of threads for better utilization given workloads with limited parallelism. We plan to explore the derivation of FlexPRET schedules from real-time constraints expressed in LF programs.

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] Edwards, S. A., and Lee, E. A. The case for the precision timed (PRET) machine. In *DAC '07: Proceedings of the 44th annual conference on Design automation* (New York, NY, USA, 2007), ACM, pp. 264–265.

[2] Lohstroh, M., et al. Actors revisited for time-critical systems. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019* (2019), ACM, pp. 152:1–152:4.

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

[4] Wilhelm, R., et al. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys. 7*, 3 (2008), 1–53.

[5] Wilson, R. Is tomorrow's embedded-systems programming language still C? [Online, accessed June 2019].

[6] 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).