Reactors: A Deterministic Model for Composable Reactive Systems*

Marten Lohstroh[1], Íñigo Íncer Romeo[1], Andrés Goens[2], Patricia Derler[3], Jeronimo Castrillon[2], Edward A. Lee[1], and Alberto Sangiovanni-Vincentelli[1]

[1] Dept. of Electrical Engineering and Computer Sciences, UC Berkeley, USA
{marten,inigo,eal,alberto}@berkeley.edu
[2] Chair for Compiler Construction, TU Dresden, Germany
{andres.goens,jeronimo.castrillon}@tu-dresden.de
[3] National Instruments
patricia.derler@ni.com

Abstract. This paper describes a component-based concurrent model of computation for reactive systems. The components in this model, featuring ports and hierarchy, are called reactors. The model leverages a semantic notion of time, an event scheduler, and a synchronous-reactive style of communication to achieve determinism. Reactors enable a programming model that ensures determinism, unless explicitly abandoned by the programmer. We show how the coordination of reactors can safely and transparently exploit parallelism, both in shared-memory and distributed systems.

1 Introduction

In the mid-80s, David Harel and Amir Pnueli introduced the notion of *reactive* systems as those systems which maintain an ongoing interaction with their environments [28]. Arguing that a suitable decomposition mechanism for the development of complex reactive systems was lacking at the time, Harel proposed Statecharts [29], a formalism based on state machines. State machines, however, must keep track of a global state, a demand too stringent for programming today's distributed systems.

More recently, the term "reactive system" has been adopted by the reactive programming community, which is focused on building flexible, loosely-coupled, and scalable systems [34]. Central to the so-called reactive design patterns is the idea of decomposing systems into non-blocking, asynchronous tasks that

* The work in this paper was supported in part by the National Science Foundation (NSF), awards #CNS-1836601 (Reconciling Safety with the Internet) and #CNS-1739816 (Quantitative Contract-Based Synthesis and Verification for CPS Security) and the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by 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).

communicate via messages (or events). The ideas expressed in the Reactive Manifesto [10] can largely be seen as a revival of the concepts behind the Actor model by Hewitt and Agha [30]. While scalability, resilience, elasticity, and responsiveness—all tenets of the manifesto—are clearly important, the gains in these dimensions come at the loss of testability due to the admittance of nondeterminism. This is a rather high price to pay, because systematic testing is still the single most common technique for ensuring the correctness of software. We argue that the goals of reactive programming can also be achieved without adopting a nondeterministic programming model, with the advantage of maintaining the ability to reliably reproduce and debug potential problems.

In this paper we describe *reactors*, a model of computation that offers many desirable properties for designing and programming reactive systems. Our model is deterministic by construction while allowing for nondeterminism that is introduced explicitly. Unlike most reactive design patterns and programming models, timing is a fundamental element in the semantics of reactors. As such, reactors are also particularly suited for specifying real-time requirements in software. The carefully-coordinated relationship between logical and physical time during the execution of reactors allows for the detection and handling of timing violations. By the same token, safe-to-progress analysis (as it is known from Ptides [61,21] and Google Spanner [16]) can be leveraged to maintain a deterministic semantics between reactors distributed across networked nodes. Similar to reactive programming languages, the execution of reactors is governed by a precedence graph. More generally, this graph is a partial order which exposes parallelism that can be exploited at runtime. Finally, the interfaces of reactors readily expose dependencies, allowing their functionality to be treated as a black box, and opening up the possibility for a polyglot language design.

Reactors were first proposed in [46] and have been discussed in subsequent papers [44,45]. The main contribution of this paper is to provide a formal description of reactors as well as algorithmic descriptions of the key building blocks required for implementing a reactor runtime system.

1.1 The case for determinism

It may be argued that rapid recovery at run time to ensure correct behavior is preferable to statically asserting properties of software. After all, hardware failures, power outages, and other external influences can break the very assumptions in the programming model that imply determinism. While this is true, the dramatic success of software is squarely due to the high probability of hardware behaving deterministically, so the value of such a deterministic model is undeniable. Moreover, particularly in a cyber-physical system (CPS), the cost of recovery may be unacceptable, as the effects of unintended behavior could be irreversible–even disastrous. And even when recovery from unexpected errors is necessary, it is helpful to test those scenarios to assure that they are handled correctly.

We can look at Toyota's unintended acceleration case to underscore the impact of nondeterminism on testability. In the early 2000s, there were a number of

serious car accidents involving Toyota vehicles that appeared to suffer from unintended acceleration. The US Department of Transportation contracted NASA to study Toyota software to determine whether software was capable of causing unintended acceleration. The study [52] was unable to find a "smoking gun," but concluded that the software was "untestable," making it impossible to rule out the possibility of unintended acceleration [33]. The software used a style of design that tolerates a seemingly innocuous form of nondeterminism. Specifically, state variables representing, for example, the most recent readings from a sensor, were accessed unguardedly by a multiplicity of threads. The spirit of this style of programming is to favor reactivity over consistency; the trade-off that is also central to the reactive programming paradigm. This programming style, however, renders software untestable, because, given any fixed set of inputs, the number of possible behaviors is vast.

A programming model can meaningfully limit the kinds of behaviors that a programmer can express. While weakening the constraints of a programming model can be useful for very specific optimization purposes, by far, most programmers will greatly benefit from a stricter rule set that facilitates the design of systems that will behave correctly and predictably [38]. Lightweight formal methods, such as type checking and static analysis, are well known to greatly reduce programming faults, for instance. The goal of reactors is to impose restrictions on the set of allowable behaviors without being *too* restrictive. For instance, the reactor model allows mutable shared state, but only across code segments that are guaranteed to execute sequentially to ensure mutual exclusion, and must execute in a predefined order to ensure determinism, making it much easier for the programmer to reason about side effects. Similarly, reactors are coordinated so that they automatically exploit opportunities for parallel execution, but only when possible without introducing nondeterminism. This relieves the programmer of the burdensome task of performing such coordination explicitly. In essence, the programming model prevents the formulation of programs that exhibit nondeterminism accidentally; nondeterminism is allowed, but it requires the express intent of the programmer.

1.2 Outline

The paper is organized as follows. We present the concept of reactors informally (Section 2) with a motivating example. We then proceed to formally define our model (Section 3) and show how our construction achieves a deterministic, synchronous-reactive model with a modular, hierarchical structure and an inherent notion of time. In Section 4, we explain how our model is amenable to distributed execution. We discuss related work in Section 5. Finally, we conclude and discuss avenues of future work in Section 6.

2 Reactors

In our model, a *reactor* is a collection of routines, called *reactions*, which share common *state*. The anatomy of a reactor is illustrated in Figure 1. The quadri-

lateral in the top middle of the figure represents the reactor's state, with state variables s_i through s_n. Reactors can contain other reactors, connected in some topology, illustrated in the figure in the area below the reactor's state. A contained reactor has no access to its container's state, but it can be connected to its container's ports via reactions (annotated as n_p in Figure 1, where p denotes the reaction's *priority*). The priorities assigned to reactions determine the order of execution of simultaneously triggered reactions. We distinguish between logical time t, which is the time of the model, represented as *tags* (as in the tagged signal model [39]), and physical time, as it would pass on a wall clock. In our model, simultaneity is a purely logical notion. Events in the reactor model are tagged; this orders them along a logical timeline. Two events (and therefore, the reactions they trigger) are simultaneous if and only if their tags are equal.

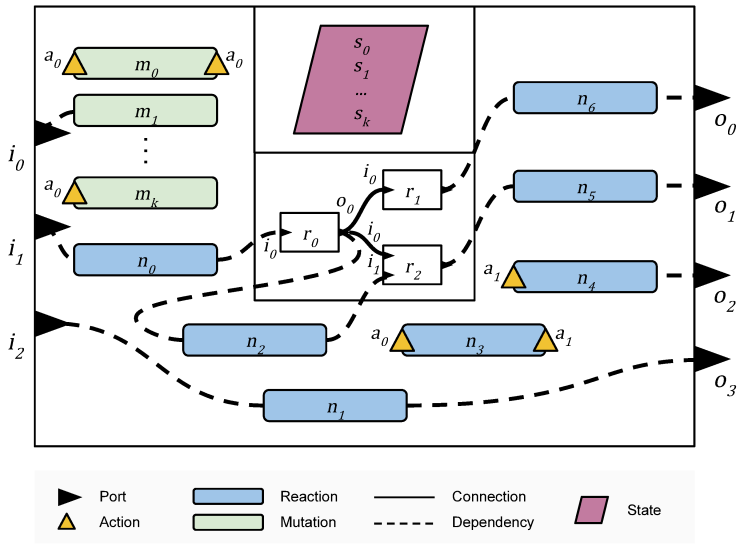

Fig. 1. Schematic representation of a reactor

The fundamental unit of execution is the reaction. Reactions are routines that may operate on the common state of the reactor and have access to a subset of the *input* and *output* ports of the reactor. Reactions may also communicate with other reactions through signals that are internal to a reactor; these are called *actions* and are depicted as the small triangles labeled a_i in Figure 1. Ports and actions are named entities that can carry *values*. In an implementation, these would likely be typed, but for the sake of simplicity we omit types from our discussion. Reactions are opaque; they can modify the shared state of a reactor and have side effects, such as reading a sensor or driving an actuator, but these effects are not captured in the model.

The rounded boxes in the top-left corner of Figure 1, annotated as m_p, where p denotes priority, are not reactions; we call them *mutations*. We make a distinction between reactions and mutations because, unlike a reaction, a mutation can modify the dependency graph of the reactor that contains it by adding connections to its topology, removing connections, and/or adding or removing its reactions. As a consequence, to preserve determinism, all mutations triggered at a given logical time t must be carried out prior to the execution of any contained reactions that are triggered at t. There are important advantages to limiting the scope of mutations to the internals of a reactor. For example, it allows mutations to be carried out without requiring any coordination with adjacent reactors. A mutation has to declare all the ports that it references. Hence, it can only establish new connections such that when this would introduce an algebraic loop, it would be detectable locally, considering only the elements contained by the reactor itself and without inspecting the broader connection topology the reactor is embedded in. In other words, hierarchy helps to contain the effects of run-time mutations.

The usage of *ports* (filled black triangles in the figure) establishes a clean separation between the functionality and composition of reactors; a reactor only references its own ports or ports of reactors it contains, not the ports of adjacent reactors. This is a key difference with Hewitt actors [31]—as featured in Akka [55], for instance—which address each other directly in a single shared address space. If, for a given tag, a reactor sets the value of an output port it has, this value will be propagated to the input ports of downstream reactors connected to it. Reactions are logically instantaneous. Logical time does not elapse during a reaction. Reactions have dependencies on input ports and antidependencies on output ports, shown as dashed edges in the figure. A reaction is not allowed to execute before all values associated with its dependencies are known (i.e., an upstream event with a tag t may not be emitted after a downstream reaction dependent on that event has already executed at t).

Just like in functional reactive programming languages, the dependency information of reactions is used to avoid so-called "glitches" (i.e., transient appearances of inconsistent data [15]) and ensure that execution unfolds in a predictable fashion. Rather than inferring dependencies from code, however, reactions must declare them, similar to how a function has to declare its arguments. This approach decouples the coordination problem from the implementation language, which led us to develop a meta language we call *Lingua Franca* (LF). This language serves the sole purpose of declaring and composing reactors, and superimposing a timing semantics on their execution. The program logic can then be written in the target language of choice. The specifics of the language and compiler toolchain are outside of the scope of this paper.

2.1 Runtime API

Reactions have access to a small set of primitives:

– LogicalTime: Returns the current *tag*;

- GET: Returns the value associated with given port/action at the current tag;
- SET: Binds given value to a given port at the current tag;
- PHYSICALTIME: Returns the last observed physical time; and
- SCHEDULE: Schedules given action with minimum delay of one *microstep*.

These primitives are the only means provided for reactors to interact with other reactors[4]. While GET and SET facilitate synchronous communication with reactions in other reactors, SCHEDULE is intended to trigger reactions within the *same* reactor, via an action. Actions can have a delay associated with them, which SCHEDULE uses to determine the tag of the resulting event. Moreover, an action must have a specified origin: *logical* or *physical*. When scheduled, an action with a logical origin (i.e., a logical action) will be scheduled relative to the last known logical time. Conversely, actions with a physical origin (i.e., physical actions) are scheduled relative to the last known physical time, a time value obtained from the platform. To avoid causality loops, logical actions are always scheduled with a minimum delay of one microstep. A microstep delay is an increment of the index in superdense time [40,47,3] with respect to the current logical time.

Like ports, actions can carry values. If more than once a particular action gets scheduled to occur at a particular time, the last set value persists. The same holds for ports. Multiple reactions could be triggered at the same logical time, and when two such reactions set the value of the same port, the earlier set value is overwritten. Because all triggered reactions within a reactor are executed in a predefined order, this semantics does not lead to nondeterminism, and it assures that each value is defined uniquely for each tag. Of course, this is only true if each port can have at most one incoming connection. This requirement has to be strictly enforced. Ports and actions that have not been set have the value *absent*. After all reactions for a given tag have been executed, the values of all ports are set to absent. In other words, ports and actions are, by default, not persistent.

The subtle interaction between logical and physical time in the reactor model establishes an interface between inherently asynchronous and nondeterministic concurrent tasks (e.g., a sensor that monitors a physical process) and deterministic computational tasks that benefit from testability and could require precise and predictable timing (e.g., to drive an actuator to influence said physical process). Rather than superimposing a deterministic world view on things that are inherently unpredictable, or, rejecting determinism entirely—thereby fundamentally compromising testability—reactors provide a model of computation that avoids this false dichotomy.

While all reactors in a reactor program share the same logical and physical clock, reactor programs can interact with one another in a distributed setting, where each program has its own logical and physical clock. The preservation of a deterministic semantics in such a setting relies on assumptions about network

[4] Primitives used by a mutation to effect changes to its container's connection topology or its reactions are not discussed here due to space limitations.

delay and clock synchronization error. In this setting, reactions must have access to PhysicalTime to check for violations of these assumptions. This is explained in more detail in Section 4.

2.2 Example: Drive-by-wire system

To illustrate how reactors behave, let us return to the Toyota example mentioned in the introduction and consider a power train implemented using reactors, illustrated using the diagram in Figure 2. It features six reactors that jointly coordinate the control of the brakes and the engine. While this example is obviously oversimplified, it features enough complexity to allow us to highlight some of the most interesting aspects of our model. Following the "accessor" pattern from [12], each reactor in the figure (represented by a rectangular box) endows a complex subsystem of the car with a simple interface that allows it to be connected to other reactors. Connections are shown as solid lines in the diagram.

Consider the LP (left pedal) reactor, in Figure 2, which is used to control the brakes. We assume that updates from the pedal are reported via an interrupt, which enables an interrupt service routine (ISR) that schedules an internal action. This internal action triggers a reaction that sets the value of the angle and on/off output ports. In order to avoid overwhelming the system, we assume that the interrupts have a minimum interarrival time. The values angle and on/off, if present, are propagated to BC (brake controller) and EC (engine controller), respectively. Notice that LP only has to set on/off at times that the pedal changes from being released to pressed and vice versa. This prevents the system from being burdened with handling insignificant events.

Let us now consider the EC reactor, which has three reactions. We interpret the number associated with each reaction as its execution priority; this way, we obtain an execution order in case both on/off and angle are present at the same logical time. The first reaction, EC.1, is triggered by on/off; it updates the state of the reactor to reflect that the brake pedal is currently pressed and sets the value of torque to zero. The second reaction is triggered by the angle input; it checks whether the brakes are applied, and if not, sets the torque output. The third and last reaction sets the value of check to trigger a reaction in the RP (right pedal) reactor, which represents the accelerator pedal. It only sets the value of check, however, if the brake pedal is known not to be pressed. This reaction is triggered by an action, which, in a naive implementation, could arrive at regularly spaced intervals. The frequency of these periodic actions, however, would have to match the maximum number of rotations per second of the crankshaft, which, under normal driving conditions, is rarely realized. Therefore, it would be more efficient to trigger the second reaction with variable intervals depending on the number of revolutions of the crank shaft.

The second reaction of RP is triggered by the check input and sets in motion some asynchronous activity that senses the angle of the accelerator pedal and writes it to the reactor's shared state. Before concluding the second reaction, an action is scheduled at the current time plus a *delay* of 2 milliseconds, to give

the ADC ample time to report its reading. The first reaction is triggered by this action and, in turn, triggers the second reaction in EC.

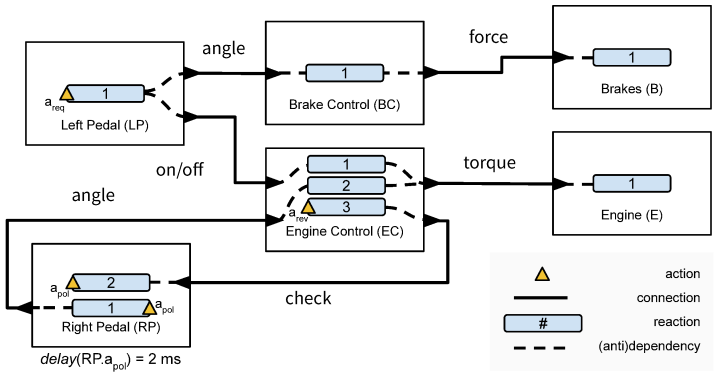

Fig. 2. Reactor that implements a simplified power train control module

The design assures that if the accelerator pedal is stuck or reports faulty readings, the car will still slow down in response to the break pedal being pressed; the engine is never allowed to apply torque when the brakes are applied. Note that this approach does not attempt to artificially eliminate nondeterminism that is intrinsic to the physical realization of the system; actions can occur sporadically, but the logic constituted by reactions *is* deterministic, and therefore, testable. The behavior of the system is relatively easy to reason about, and it is straightforward to formulate meaningful test cases to build confidence in the correctness of the implementation of the reactions.

Real-time constraints In the example in Figure 2, the first reaction of LP is triggered by an interaction with the environment; a process that reports a sensor value. The scheduling of physical action a_{req} effectively maps physical time to logical time. The event output by LP is timestamped with logical time t equal to physical time T. We can now impose a *deadline*[5] on the triggering of a reaction in downstream reactor B of, say, 5 milliseconds. This means that the force input of the brakes must be observed before or when T reaches $t+5ms$. In other words, the deadline specifies a maximum end-to-end delay in physical time between the reaction that reports the braking and the reaction that applies the brakes. The execution engine can therefore, by comparing logical timestamps with physical time, check for deadline violations at run time. Moreover, the presence of the deadline enables Earliest Deadline First (EDF) scheduling. In combination with precision-timed hardware and Worst-Case Execution Time (WCET) analysis, static guarantees could be obtained with regard to timing of reactions [35,45].

[5] Deadlines are omitted from the formalization in Section 3 and are left as future work.

3 Formalization

In this section we formalize the concept of reactors. Some of the central concepts we will introduce are described by lists of elements. In order to simplify notation, we will use the symbol for the element of a list to also denote a function that maps the list to the element corresponding to that symbol. For example, if x is defined as the list $x = (a, b)$, we reuse the symbols a and b to be functions that map x to its elements a and b, respectively. Thus, we will commonly use the notation $a(x)$, where x is a list, and a is the symbol of one of the elements in that list.

First, we need to introduce some notation. Let Σ be a set. We refer to the elements of Σ as *identifiers*. We will use identifiers to uniquely refer to various objects to be introduced. There is no need to further define the structure of identifiers.

Let $\mathfrak{V}$ be a set, which we refer to as the *set of values*. This set represents the data values exchanged between or within reactors. Similarly, we do not assume any structure in the values, i.e., reactors are untyped. We define one distinguished element in the value set: $\varepsilon \in \mathfrak{V}$ is called the *absent value*.

As motivated in Section 2, a reactor is a composite of various objects. Some of these objects have roles which are tightly intertwined with the model of computation in which reactors operate. This model of computation is the *discrete event* model. In discrete event systems, the execution of a program occurs at given *tags*. These tags belong to a denumerable and totally ordered set.

3.1 Notions of time

Our model uses a superdense representation of time (see [40,47]). Each tag is denoted by a pair, of which the first element is a *time value*—an integer representation of time in some predefined unit (e.g., milliseconds or nanoseconds)—and the second element denotes a *microstep index*. Two events are logically simultaneous if and only if their tags are equal. Formally, the set of tags, $\mathbb{T}$ of the reactor execution model is $\mathbb{T} = \mathbb{N}^2$, where $\mathbb{N}$ is the set of natural numbers. We define a total order on $\mathbb{T}$ lexicographically: if $(a, b), (a', b') \in \mathbb{T}$, we say that $(a, b) < (a', b')$ if and only if $(a < a') \vee (a = a' \wedge b < b')$. $\mathbb{T}$ has an addition operation that operates element-wise. Using an integer representation for time ensures that addition is associative, which is not necessarily the case when using floating-point representations [17]. We define a function TIMEVAL on tags which extracts the time value: let $(a, b) \in \mathbb{T}$, then $\text{TIMEVAL}((a, b)) = a$.

Events are used to exchange messages between reactors.

Definition 1 (Event). *An event e is defined as a list $e = (\mathfrak{t}, \mathfrak{v}, \mathfrak{g})$, where $\mathfrak{t} \in \Sigma$ is called the **event trigger**, $\mathfrak{v} \in \mathfrak{V}$ the **trigger value**, and $\mathfrak{g} \in \mathbb{T}$ the **event tag**. Events inherit an order from their tags. If e and e' are events, we say that $e < e'$ if and only if $\mathfrak{g}(e) < \mathfrak{g}(e')$.*

Definition 2 (Event queue). *We define the **event queue** $\mathcal{Q}_E$ as a set of events ordered by their tags.*

This model uses two distinct notions of time: *logical time* and *physical time*.

Definition 3 (Logical time). *Logical time is a monotonically increasing sequence of tags of the form (a, b), where a is referred to as the **time value** and to b as the **microstep index**.*

Definition 4 (Physical time). *Physical time refers to a time value that is obtained from a clock on the execution platform.*

Remark 1 (Time units). The time values of logical time and physical time must be given in some unit of measurement. In order to meaningfully relate two time values, their units must be the same. Whenever we omit units in expressions that relate time values, we simply assume the units match. Microstep indices, on the other hand, are unitless.

3.2 Reactors

We now proceed to define reactors. Note that reactors contain *reactions* and *mutations*. We first discuss reactors to clarify how the domain of constituents of a reaction or mutation is determined by the containing reactor.

Definition 5 (Priority set). *Let $\mathbb{Z}$ be the set of integer numbers, $\mathbb{Z}^+$ the set of integers larger than zero, $\mathbb{Z}^-$ the set of integers smaller than zero, and $*$ a symbol which is not an integer. The priority set, $\mathcal{P}$, is given by $\mathcal{P} = \mathbb{Z}^- \cup \mathbb{Z}^+ \cup \{*\}$. The set $\mathcal{P}$ has a partial order given by the order in $\mathbb{Z}$ extended with $* \leq *$ and $p < *$ for all $p \in \mathbb{Z}^-$.*

The use of $*$ is to allow particular reactions to be executed in parallel if they do not touch the reactor's state. For instance, reactions n_0, n_5, and n_6 in Figure 1 would qualify as such if their only purpose is to relay values between ports.

Definition 6 (Action). *An action is a list $a = (x, d, \mathfrak{o})$, where $x \in \Sigma$ is the **action identifier**, $d \in \mathbb{N}$ is the **delay** of the action, and $\mathfrak{o} \in \mathfrak{O}$ is the **origin** of the action. We use the notation $d(x)$ to refer to the delay of a, and $\mathfrak{o}(x)$ to refer to its origin. As we will see, actions belong to exactly one reactor. If A is a set of actions, we will also let $x(A)$ denote the set of identifiers of each action in A.*

When a reaction or mutation schedules an event for an action, this event will have a tag that includes the action delay plus either the current logical time or the current physical time, depending on whether the action's origin is *logical* or *physical*, respectively. This is described in more detail in Algorithm 2 in Section 3.5.

Definition 7 (Reactor). *A reactor r is a list $r = (I, O, A, S, \mathcal{N}, \mathcal{M}, \mathcal{R}, \mathcal{G}, P, \bullet, \diamond)$, where*

1. *$I \subseteq \Sigma$ is a set of **inputs**,*

2. $O \subseteq \Sigma$ *a set of* ***outputs****,*
3. $A \subseteq \Sigma \times \mathbb{N} \times \mathfrak{O}$ *a set of* ***actions****,*
4. $S \subseteq \Sigma$ *a set of* ***state variables****,*
5. $\mathcal{N}$ *a set of* ***reactions****,*
6. $\mathcal{M}$ *a set of* ***mutations****,*
7. $\mathcal{R}$ *a set of* ***contained reactors****,*
8. $\mathcal{G} \subseteq \left(\bigcup_{r \in \mathcal{R}} O(r)\right) \times \left(\bigcup_{r \in \mathcal{R}} I(r)\right)$ *a* ***topology graph****,*
9. $P : \mathcal{N} \cup \mathcal{M} \to \mathcal{P}$ *the* ***priority function****, and*
10. $\bullet, \diamond \in A(r)$ *actions called* ***initialization*** *and* ***termination****, respectively.*

Given two reactors r and r', the sets $I(r)$, $O(r)$, $x(A(r))$, $S(r)$, $I(r')$, $O(r')$, $x(A(r'))$, and $S(r')$ are all pairwise disjoint. Similarly, the sets $\mathcal{R}(r)$ and $\mathcal{R}(r')$ are disjoint, and so are the sets $\mathcal{N}(r)$ and $\mathcal{N}(r')$ and $\mathcal{M}(r)$ and $\mathcal{M}(r')$.

Remark 2 (Hierarchy). We define an *atomic* reactor as above, with an empty contained reactor set and empty topology graph, and we call these degree-0 reactors. Then, for $n \geq 1$ we define a reactor of degree n as a reactor with a set $\mathcal{R}$ of reactors of degree at most $n - 1$ and a corresponding topology set. Moreover, the reactor set of a degree-n reactor contains at least one reactor of degree $n - 1$.

Reactors use their inputs and outputs to communicate with other reactors. Reactions and mutations can schedule events for actions in order to trigger the execution of other reactions or mutations contained in the same reactor.

Reactors can be built up hierarchically. As such, the reactor set lists the reactors contained by a reactor. The reaction and mutation sets list the reactions and mutations, respectively, contained by a reactor. The topology graph specifies how the reactors contained in a reactor are connected to each other. This graph consists of pairs (o, i), where o and i are the output and input of two reactors, respectively. If $(o, i), (o', i')$, are two different elements of $\mathcal{G}$, then $i \neq i'$; that is, inputs are connected to at most one output. The priority function plays a role in the concurrent execution of reactions and mutations and is discussed in Section 3.8. Initialization and termination actions are discussed in Section 3.7.

Let us consider the constituents of the reactor shown in Figure 1: $I = \{i_i\}_{i=0}^{2}$, $O = \{o_i\}_{i=0}^{3}$, $A = \{a_i\}_{i=0}^{1} \cup \{\bullet, \diamond\}$, $\mathcal{N} = \{n_i\}_{i=0}^{6}$, $\mathcal{M} = \{m_i\}_{i=0}^{k}$, and $\mathcal{R} = \{r_i\}_{i=0}^{2}$. The priorities of the reactions and mutations shown in the figure are equal to their respective subindices. The topology graph is the set of pairs which indicate connections from the outputs to the inputs of contained reactors. In the figure, these pairs are $(r_0.o_0, r_1.i_0)$ and $(r_0.o_0, r_2.i_0)$.

3.3 Reactions

We now discuss the elements that carry out computation in the reactor model. These are called reactions. First, we define a function to navigate the reactor hierarchy:

Definition 8 (Container function). *The container function C maps a reactor r to the reactor which contains it. The function returns $\top$ (pronounced "top") if no reactor contains r. Since the sets $\mathcal{R}(r), \mathcal{R}(r')$ are disjoint for $r \neq r'$, C is well-defined. Let r be a reactor. If $C(r) = \top$, we say that r is* ***top-level****. We also define the container function for reactions and mutations: let n be a reaction; then $C(n)$ yields the reactor r such that $n \in \mathcal{N}(r)$. The same applies to mutations. Finally, we define the container function for inputs, outputs, and action identifiers: let i, o, and a be an input, output, and action, respectively, of three reactors r, r', and r''. Then $C(i) = r$ if and only if $i \in I(r)$, $C(o) = r'$ if and only if $o \in O(r')$, and $C(x(a)) = r''$ if and only if $a \in A(r'')$. Similarly, the function C is well-defined here since all the relevant sets are pairwise disjoint for two distinct reactors.*

With this function in place, we state the definitions of reactions:

Definition 9 (Reaction). *A reaction n is defined as $n = (D, \mathcal{T}, B, D^{\vee}, H)$, where*

1. $D \subseteq I(C(n)) \cup \bigcup_{r \in \mathcal{R}(C(n))} O(r)$ is a set of ***dependencies****, identifiers on which the reaction depends in order to execute;*
2. $\mathcal{T} \subseteq D \cup x(A(C(n)))$ is a set of ***triggers****, identifiers whose events cause the execution of the reaction's body;*
3. B is the ***body*** *of the reaction (e.g., executable code);*
4. $D^{\vee} \subseteq O(C(n)) \cup \bigcup_{r \in \mathcal{R}(C(n))} I(r)$ is the set of ***antidependencies****, identifiers for which the reaction can produce events at the current logical time; and*
5. $H \subseteq x(A(C(n)))$ is the set of ***schedulable actions****, actions for which n can generate events.*

3.4 Mutations

Now we introduce the concept of a mutation. These are used to modify the internal structure of a reactor by connecting and disconnecting ports. Ports that a mutation declares as dependencies are the only sources that it can establish connections from. Ports that it declares as antidependencies are the only destinations that it can establish connections to. While mutations give reactors the ability to dynamically reconfigure their internal topology, the above constraints prevent a reactor from introducing dependencies between its ports of which its container is not already aware.

Definition 10 (Mutation). *A mutation m is defined as $m = (D, \mathcal{T}, B, D^{\vee}, H)$, where*

1. $D \subseteq I(C(m))$ is a set of ***dependencies****, identifiers on which the mutation depends in order to execute, and the only sources from which the mutation can establish connections;*
2. $\mathcal{T} \subseteq D \cup x(A(C(m)))$ is a set of ***triggers****, identifiers whose events cause the execution of the mutation's body;*

*3. B is the **body** of the mutation (i.e., executable code);*
*4. $D^{\vee} \subseteq O(C(m)) \cup \bigcup_{r \in \mathcal{R}(C(m))} I(r)$ is the set of **antidependencies**, identifiers for which the mutation can produce events at the current logical time, and the only destinations to which the mutation can establish connections; and*
*5. $H \subseteq x(A(C(m))) \cup \bigcup_{r \in \mathcal{R}(C(m))} x(\{\bullet(r), \diamond(r)\})$ is the set of **schedulable actions**, actions for which m can generate events.*

Reactions and mutations differ as follows:

– Mutations can modify the reactor topology; reactions cannot.
– A mutation can schedule initialization and termination actions for reactors that its container contains.
– The outputs of contained reactors are allowed in the dependencies of a reaction, but not in the dependencies of a mutation. This is important because, in contrast to reactions, mutations have the capability of scheduling initialization actions, which do not incur a microstep delay. Disallowing outputs of contained reactors rules out the introduction of undetectable causality loops.

Appendix A summarizes all definitions we introduce.

3.5 Event generation

We will find it convenient to have available functions that return the reactions which depend on the given input, and which are antidependent on the given output. We find no reason to introduce new notation. Thus, we define the maps

$$\mathcal{N}(i) = \{n \in \mathcal{N}(C(i)) \mid i \in D(n)\} \text{ and}$$
$$\mathcal{N}(o) = \{n \in \mathcal{N}(C(o)) \mid o \in D^{\vee}(n)\}.$$

We define $\mathcal{M}(i)$ and $\mathcal{M}(o)$ for mutations in a similar manner. Moreover, given an identifier $\mathfrak{t}$, we will identify the reactions and mutations that are triggered by $\mathfrak{t}$. We define

$$\mathcal{T}(\mathfrak{t}) = \{k \in \mathcal{N}(C(\mathfrak{t})) \cup \mathcal{M}(C(\mathfrak{t})) \mid \mathfrak{t} \in \mathcal{T}(k)\}.$$

We now discuss how events are created. The body of a reaction is a container for application code in the reactor framework. Let n be a reaction. Then the body $B(n)$ of this reaction is allowed to run two functions that affect the execution environment: Schedule and Set.

A reaction can only execute Set on its antidependencies. The execution of Set in the body of a reaction propagates the set value to downstream ports and adds triggered reactions to $\mathcal{Q}_R$, the set of reactions to be executed at the current logical time. Set is shown in Algorithm 1.

A reaction can only call Schedule on its set of schedulable actions. The event created on a call to schedule is shown in Algorithm 2. The algorithm shows that reactions can add an additional delay to the delay of a schedulable action upon scheduling. Note also that Schedule can be called synchronously,

Algorithm 1 Propagate values to downstream ports

```
1: procedure SET(port, value)
2:     WRITEVALUE(port, value)
3:     reactionsAndMutations ← 𝒯(port)
4:     r ← C(C(port))
5:     topology ← 𝒢(r)
6:     for all (o, i) ∈ topology do
7:         if port = o then
8:             WRITEVALUE(i, value)
9:             reactionsAndMutations ← reactionsAndMutations ∪ 𝒯(i)
10:        end if
11:    end for
12:    𝒬_R ← 𝒬_R ∪ reactionsAndMutations
13: end procedure
```

from a reaction, but also asynchronously, from another thread of execution. Mutual exclusion between concurrent calls to SCHEDULE is achieved via locking. The same mutex is also used in NEXT, the function that drives the execution of triggered reactions (see Section 3.8). The mutex protects the event queue $\mathcal{Q}_E$, as well as the variable t that holds the current logical time, from data races.

Algorithm 2 Schedule an action

```
1: procedure SCHEDULE(a, additionalDelay, value)
2:     interval ← d(a) + additionalDelay
3:     LOCK(mutex)        ▷ Mutual exclusivity with concurrent SCHEDULE and NEXT
4:     if 𝔬(a) = Physical then
5:         tag ← (PHYSICALTIME() + interval, 0)
6:     else
7:         if interval = 0 and a ≠ •(C(x(a))) then
8:             tag ← LOGICALTIME() + (0, 1)                    ▷ Add microstep delay
9:         else
10:            tag ← (TIMEVAL(LOGICALTIME()), 0) + (interval, 0)
11:        end if
12:    end if
13:    e ← (x(a), value, tag)
14:    𝒬_E ← 𝒬_E \ {e′ ∈ 𝒬_E | 𝔱(e′) = 𝔱(e) ∧ 𝔤(e′) = 𝔤(e)}    ▷ Overwrite if already set
15:    𝒬_E ← 𝒬_E ∪ {e}
16:    UNLOCK(mutex)                                           ▷ Release mutex
17: end procedure
```

3.6 Reaction precedence

During the execution of a reactor, there may be multiple events scheduled at the same logical time. These events may trigger multiple reactions and mutations.

In what order can these reactions and mutations execute? We arrange reactions in a partial order based on their dependencies and priority with respect to other reactions within a reactor. Let k and k' be mutations or reactions. We say that $k \prec k'$ if k' has a dependency on an antidependency of k or if $C(k) = C(k') \wedge (P(k) < P(k'))$.

The analysis of dependencies excludes actions because actions (with exception of $\bullet$) are always scheduled at least one microstep time unit into the future.

Example: The precedence graph obtained from our example in Figure 2 is shown in Figure 3. Notice that $\mathsf{EC.1} \prec \mathsf{EC.2} \prec \mathsf{EC.3}$ and $\mathsf{RP.1} \prec \mathsf{RP.2}$ are due to reaction priority; the other precedence relations between reactions are due to their dependencies and antidepenencies on ports and the connections between those ports.

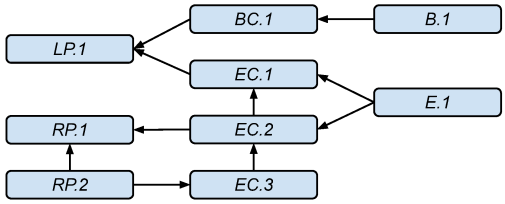

Fig. 3. Precedence graph implied by the reactor topology in Figure 2

Definition 11 (Precedence function). *Let r be a reactor. The precedence function $\gamma(r)$ returns a graph whose vertices are all reactions and mutations contained in the hierarchy of r and whose directed edges denote dependencies between vertices. The precedence function is computed according to Algorithm 3.*

These are the steps of the algorithm:

– L2. Make the vertices and edges of the precedence graphs of the constituent reactors of r part of the graph of r. We define the union of graphs to operate element-wise (i.e., on the vertex sets and edge sets).
– L3. Make the mutations and reactions of r vertices of the graph.
– L4. Use the topology to connect the reactions and mutations of contained reactors.
– L5-6. Connect the reactions of r to the reactions and mutations of the constituent reactors of r. Note that the functions $\mathcal{N}$ and $\mathcal{M}$, when applied to inputs and outputs, return the reactions and mutations which list that input as a dependency or the reactions and mutations that list that output as an antidependency.
– L7. For all reactions and mutations of this reactor, we add an edge to the graph between two reactions or mutations when the priority of one is smaller than the priority of the other.
– L8. Make all reactions and mutations of the contained reactors dependent on the mutations of the container reactor. This is necessary because mutations can

change the operation of the container reactor and can schedule the initialization action of the contained reactors with zero delay.

After computing the precedence graph using Algorithm 3, the graph must be checked for directed cycles. Cyclic precedence graphs must be rejected, as they represent algebraic loops; we do not handle them.

Algorithm 3 Construct precedence graph

1: **procedure** $\gamma(r)$
2: $\quad (V, E) \leftarrow \bigcup_{r' \in \mathcal{R}(r)} \gamma(r')$
3: $\quad V \leftarrow V \cup \mathcal{N}(r) \cup \mathcal{M}(r)$
4: $\quad E \leftarrow E \cup \bigcup_{(o,i) \in \mathcal{G}(r)} (\mathcal{N}(i) \cup \mathcal{M}(i)) \times (\mathcal{N}(o) \cup \mathcal{M}(o))$
5: $\quad E \leftarrow E \cup \bigcup_{\substack{n \in \mathcal{N}(r) \\ i \in D^{\vee}(n) \setminus O(r)}} (\mathcal{N}(i) \cup \mathcal{M}(i)) \times \{n\}$
6: $\quad E \leftarrow E \cup \bigcup_{\substack{n \in \mathcal{N}(r) \\ o \in D(n) \setminus I(r)}} \{n\} \times (\mathcal{N}(o) \cup \mathcal{M}(o))$
7: $\quad E \leftarrow E \cup \bigcup_{k,k' \in \mathcal{N}(r) \cup \mathcal{M}(r)} \{(k, k') \mid P(k') < P(k)\}$
8: $\quad E \leftarrow E \cup \left(\bigcup_{r' \in \mathcal{R}(r)} \mathcal{N}(r') \cup \mathcal{M}(r') \right) \times \mathcal{M}(r)$
9: $\quad$ **return** (V, E)
10: **end procedure**

3.7 Initialization and termination

All reactors have a special initialization action $\bullet$. At the start of executing a reactor program, the execution environment generates one event at tag $(T, 0)$ for the initialization action of the top-level reactor. Every reactor contains a mutation that is triggered by that reactor's initialization action; this mutation initializes that reactor and schedules an event with no microstep delay on the initialization actions of all reactors that its container reactor contains.

Reactors also have a special action $\diamond$, called termination. Reactors have the ability to schedule $\bullet$ and $\diamond$ actions of their contained reactors. Upon processing a termination action (which implies the need for the existence of a reaction or mutation which is triggered by the termination action), a reactor can forward that action to its contained reactors in order for the hierarchy to terminate safely. The $\diamond$ action is scheduled *with* a microstep delay, to allow any ongoing reactions to conclude before termination is set into motion.

3.8 Execution

The execution of reactors is based on a discrete-event model of computation that guarantees determinacy, a property that can be proven by showing the existence

Algorithm 4 Process events for the next tag

```
1: procedure NEXT()
2:     LOCK(mutex)                          ▷ Mutual exclusivity with concurrent SCHEDULE
3:     if Q_E = ∅ then return
4:     end if
5:     while True do
6:         T ← PHYSICALTIME()
7:         t_next ← g(PEEK(Q_E))            ▷ Obtain the tag of the first-in-line event
8:         if T ≥ t_next then
9:             break
10:        else                  ▷ Wait until Q_E changes or physical time matches tag
11:            TIMEDWAITFOREVENTQUEUECHANGE(TIMEVAL(t_next))
12:        end if
13:    end while
14:    t ← t_next                                              ▷ Advance logical time
15:    CLEARALL()                                  ▷ Clear all inputs, outputs, actions
16:    Q_R, doneSet, execSet ← ∅, ∅, ∅
17:    E ← {e ∈ Q_E | g(e) = t}                    ▷ Gather events for current time t
18:    Q_E ← Q_E \ E
19:    UNLOCK(mutex)                                                ▷ Release mutex
20:    for all e ∈ E do
21:        WRITEVALUE(t(e), v(e))        ▷ Set the value associated with identifier t(e)
22:    end for
23:    Q_R ← ⋃_{e∈E} T(t(e))              ▷ Reactions and mutations triggered by events
24:    repeat
25:        for all k ∈ execSet do
26:            if ISDONE(k) then          ▷ Check whether executing element is done
27:                doneSet ← doneSet ∪ {k}
28:                execSet ← execSet \ {k}
29:            end if
30:        end for
31:        if Q_R ≠ ∅ then                          ▷ Execute something, if possible
32:            if THREADISAVAILABLE() then
33:                P ← Q_R ∪ execSet
34:                readyForExec ← {p ∈ P | ∄p' ∈ P. p' < p}
35:                readyForExec ← readyForExec \ execSet
36:                if readyForExec ≠ ∅ then
37:                    k ← SELECT(readyForExec)
38:                    execSet, Q_R ← execSet ∪ {k}, Q_R \ {k}
39:                    RUNINTHREAD(k)
40:                else
41:                    WAITUNTILNUMBEROFIDLETHREADSHASINCREASED()
42:                end if
43:            else
44:                WAITUNTILTHREADHASBECOMEAVAILABLE()
45:            end if
46:        else
47:            if execSet ≠ ∅ then
48:                WAITUNTILNUMBEROFIDLETHREADSHASINCREASED()
49:            end if
50:        end if
51:    until Q_R ∪ execSet = ∅
52: end procedure
```

of unique fixed points over generalized metric spaces given that the precedence graph that governs the execution (see Section 3.6) contains no directed cycles [43,48]. The execution environment keeps a notion of a global event queue $\mathcal{Q}_E$ that tracks events scheduled to occur in the future, and of a reaction queue $\mathcal{Q}_R$ that sorts reactions to be executed at the current logical time by precedence. While the event loop can be implemented in a single thread, the algorithms discussed in this section assume a multi-threaded implementation. A single mutex lock is used to guarantee thread-safe operation on the only two shared data structures: t and $\mathcal{Q}_E$. A major advantage of this design is that the use of a single lock ensures deadlock-freedom. At the beginning of execution, logical time starts at a value of $t = (T, 0)$, and it can only increase as execution progresses. Logical time increases when there are no further reactions to be executed and there are one or more events in $\mathcal{Q}_E$ with a tag greater that has a time value greater than or equal to the current physical time T.

Algorithm 4 shows how the code of reactions and mutations is executed. The algorithm proceeds as follows:

– L5-13. Determine what the next logical time should be, based on the event that is currently on top of $\mathcal{Q}_E$, and wait for physical time to match the time value of the tag. The procedure TIMEDWAITFOREVENTQUEUECHANGE blocks until either the event queue was modified or the specified physical time was reached, whichever comes first. TIMEDWAITFOREVENTQUEUECHANGE is expected to release the mutex and reacquire it after receiving a signal that an event has been added to $\mathcal{Q}_E$. This allows concurrent invocations of SCHEDULE to proceed while NEXT is waiting. In an implementation based on POSIX threads, `pthread_cond_timedwait` could be used for this.
– L14. Advance logical time to match the smallest tag currently in $\mathcal{Q}_E$.
– L15. Set the values of all ports and actions to ε.
– L17. Obtain events to process at the current logical time.
– L19. Release the mutex, allowing concurrent calls to SCHEDULE to proceed.
– L20-22. Set triggers according to the value of the event.
– L23. Obtain all reactions and mutations triggered by any of the events with a tag equal to the current logical time and insert them into $\mathcal{Q}_R$.
– L24-30. If a reaction or mutation that has been under execution is done, move that reaction or mutation to doneSet and remove it from execSet.
– L32-39. The routine THREADISAVAILABLE reports whether the runtime system has a thread available for executing the selected reaction of mutation. If this is the case, on L34-35, select one reaction or mutation from the set of minimal elements of items which are either under execution or pending, excepting, naturally, the set of executing reactions or mutations. It is ensured that no reactions or mutations that precede a ready reaction or mutation in the precedence graph end up executing concurrently with it. Note that the computation of the minimal elements uses the order on reactions and mutations defined in Section 3.6.
– L41. If all pending tasks have dependencies on currently-executing tasks, wait until one of the currently-executing tasks concludes, freeing up a thread. With POSIX threads, WAITUNTILNUMBEROFIDLETHREADSHASINCREASED could be implemented using `pthread_cond_wait`.

– L44. If there are pending tasks, but the runtime system does not have resources to accept a new task, wait until it can accept a new task. Again, `pthread_cond_wait` could be used to implement the wait.
– L48. If there are no pending tasks, but there are tasks currently in execution, wait until at least one of the tasks under execution finishes.
– L51. We iterate the loop L24-51 until there remain no reactions or mutations to be executed, and there are none currently under execution.

4 Distributed Execution of Reactors

We will now describe how reactor programs are executed when distributed over multiple nodes communicating over network. Many of the concepts around distributed execution have been introduced in prior work on Ptides [62] and are applied to reactors here. We can use Ptides to preserve the deterministic semantics of reactors across distributed reactor programs, which requires us to make some assumptions about our system. Each node in the distributed system maintains its own event queue and contains a clock that monitors and keeps track of the passing of physical time. The clocks are synchronized across nodes with a known bound E on the clock synchronization error. Sending messages between distributed nodes takes time, but we assume a known upper bound on the network delay L between any two nodes in the network.

Let us consider the example in Figure 2 and assume the reactors are distributed across multiple engine control units (ECUs). The reactor network is split up into multiple, distributed reactor networks. While automotive networks often do have provisions for deterministic communication, many networks do not guarantee in-order processing of messages, thus potentially causing the receipt of messages out of order. We can also envision an extension of this system with car-to-car communication to enable safe lane-switching, platooning, intersection management, or emergency slow down. Networks used for such communication are typically not giving any guarantees on the order of transfer of messages.

Similar to sensing and actuation wrapped in reactions, network communication is performed in the body of reactions. A network sending reaction must combine the event value together with the current logical time t and implement the network transmission. To ensure timely sending of network messages, a deadline on the network sending the reaction is required. Note that a deadline > 0 increases the delay on a path between sensors and actuators.

Just like a network sending a reaction, a reaction receiving a message from the network implements network communication in the body of the reaction. We assume an interrupt upon receipt of the network that triggers an action, which, in turn, triggers the network receiving reaction. This reaction unpacks the timestamp t_m in the message and uses it to determine when it is safe to process the message. A network receiver in this programming model must ensure that messages are forwarded to other reactors in logical timestamp order. A network receiver that receives a message m at physical time T with timestamp t_m cannot release the message until $t_m + E + L$ to ensure that no other messages are in

the network with an earlier timestamp. When physical time matches or exceeds $t_m + E + L$, the message is safe to process. A network receiving reaction will therefore schedule an action a with `additionalDelay` $= t_m + E + L - t_c$, where t_c is the current logical time on the node. A reaction triggered by a will release the message into the local reactor program.

A violation of the assumptions of clock synchronization error E or network delay L is detected if a network receiver gets a message with timestamp t at physical time T with $T > t + E + L$ (see Figure 4). Once such an error is detected, the mitigation is application dependent, ranging from ignoring such erroneous network messages to an immediate stop of the program. While we do not discuss strategies for dealing with such an error, we want to stress that the strength of this programming model is in the ability to detect such errors.

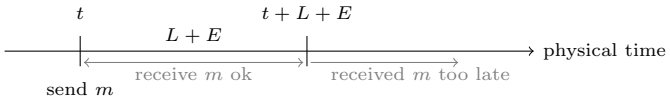

Fig. 4. Message exchange between distributed reactors

By relating physical time to logical time at sensors, actuators and network interfaces, deterministic behavior is implemented without the need for a central coordinator. The analysis of whether a distributed reactor program can be implemented on a given set of nodes is performed for each node individually, by treating network interfaces like sensors and actuators.

5 Related Work

The Actor model by Hewitt and Agha [31,1] can be considered the basis for reactors. In it, actors execute concurrently and communicate via asynchronously passed messages, with no guarantees on the order or timing of message arrival. Implementations can be found in several modern languages and software libraries, most notably Erlang [2] by Ericsson, Scala actors [27], Akka [11], and Ray [50]. The messaging is address-based, and an actor can send messages to any other actor just using its address, including actors it creates. This flexibility can be leveraged to make distributed systems more resilient. Dataflow models [19,9,36] and process networks [32,37] can be seen as subsets of actor models with deterministic semantics that allows for explicit nondeterminism. Stemming from the embedded systems community, fixed graph topologies in these models enable improved static analysis and optimization [53].

The reactive programming community is concerned with developing event-driven and interactive applications using a wide array of software technologies ranging from programming frameworks like ReactiveX [49], Akka [11,60], and Reactors.IO [54] to language-level constructs like event loops [58], futures [5],

promises [23], and reactive extensions [49]. For a more comprehensive survey on reactive programming techniques, see [4]. Writing software for reactive systems is difficult when the control flow of a program is driven by external events not under the control of the programmer, since the conventional imperative programming paradigm cannot be used. A major goal of reactive programming approaches is providing abstractions to express programs as reactions to external events (the observer pattern) and abstracting away the flow of time. Reactors have the same goal, but instead make use of synchronized time to coordinate such that their reactions yield predictable results.

In many reactive programming frameworks, futures are used to promote an imperative, sequential programming style, which avoids an explicit continuation passing style (also known as "callback hell" [20]), but makes it even more confusing for the programmer when nondeterminism rears its head. Actor-based frameworks like Ray and Akka rely heavily on futures. All this programmatic support makes reactive systems especially difficult to debug [6,59]. Another significant problem with some of the frameworks, libraries or language primitives commonly used in reactive programming is that they invite programmers to break the semantics of the underlying model, mixing models and losing many of the advantages obtained from them [57].

A different class of very successful models that reactors draw from are discrete-event models. These models, common in hardware modeling and simulation, have time as a core element in their semantics. Discrete events are, by design, the model of computation underlying reactors. From a language level, our language proposition is very close to hardware description languages, like Verilog or VHDL. Noteworthy is the comparison with SystemC [42,25], and the related SpecC [24], of which reactors are particularly reminiscent.

On the software engineering side, reactors are probably closest to synchronous languages and Functional Reactive Programming (FRP). In fact, the discrete event model can be seen as a special case of the model behind synchronous languages [41]. Synchronous languages like Esterel [8], Lustre [26] and SIGNAL [7] make time an essential part of the language design. Here, discrete time ticks are purely logical and not being synchronized to real, wall clock time. This is reminiscent of the signals used in FRP languages, like Fran [22] or FrTime [15], or more modern languages like Elm [18]. Unlike reactors, FRP works with pure functions and does not deal with side effects like reading sensors or operating actuators, which are essential in cyber-physical systems. In addition, these systems typically require a central runtime, which makes a dynamic, distributed execution infeasible.

The reactive extensions [13] to AmbientTalk make this actor-based language for mobile application design into one that is very similar to reactors. In particular, it stores a topology graph and can execute distributedly, albeit without avoiding glitches [4]. Myter et al. [51] show how to avoid glitches in reactive distributed systems using distributed dependency graphs and logical clocks to timestamp values propagated through the system. Timestamps are used to decouple distributed components thus voiding the need for central coordina-

tors and, in effect, implementing a *Globally Asynchronous, Locally Synchronous* (GALS) [14] system. Myter *et al.* propose an execution runtime which guarantees that a distributed reactive system eventually reaches a consistent state. Our work shares several key ideas with this approach, such as the use of logical time and the construction of dependency graphs to circumvent the need for central coordination, although modeling and implementations choices differ considerably. In addition, our work is based on ideas presented in Ptides [62], where logical time is carefully linked to a notion of physical time that is assumed to be synchronized across nodes with a known error tolerance. This allows for an always (not just eventual) consistent state. In addition, we can now reason about end-to-end delays and timing violations, which can help detect errors in the assumptions about the system or the execution behavior of a system.

6 Conclusions

Reactors are software components that borrow concepts from actors, dataflow models, synchronous-reactive models, discrete event systems, object-oriented programming, and reactive programming. They promote modularity through the use of ports, use hierarchy to preserve locality of causality effects, and provide a clean interface between asynchronous tasks and reactive programs without compromising the ability to obtain deterministic reactions to sporadic inputs. This makes reactors particularly well suited as a programming model for implementing cyber-physical systems, and more broadly, reactive systems that are expected to deliver predictable, analyzable, and testable behavior.

We have shown how the reactor execution model takes advantage of concurrency that is naturally exposed in reactor programs; we leave performance benchmarks, as well as analyses of different scheduling policies and a more thorough discussion of deadlines and runtime mutations as future work. We are currently developing a compiler tool chain that takes reactor definitions and compositions written in the Lingua Franca meta-language and transforms them into executable target code. Among features we intend to develop for this language are declarative primitives for the orchestration of distributed reactor programs, runtime mutations based on state machines, and real-time scheduling analysis for precision-timed hardware platforms like Patmos [56] and FlexPRET [63].

Acknowledgement

The authors thank the anonymous reviewers for their perceptive feedback on an earlier version of this paper.

References

1. Agha, G.: ACTORS: A Model of Concurrent Computation in Distributed Systems. The MIT Press Series in Artificial Intelligence, MIT Press, Cambridge, MA (1986)

2. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent programming in Erlang. Prentice Hall, second edn. (1996)
3. Bai, Y.: Desynchronization: From macro-step to micro-step. In: 2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE). pp. 1–10 (Oct 2018)
4. Bainomugisha, E., Carreton, A.L., Cutsem, T.v., Mostinckx, S., Meuter, W.d.: A survey on reactive programming. ACM Computing Surveys (CSUR) **45**(4), 52 (2013)
5. Baker Jr, H.C., Hewitt, C.: The incremental garbage collection of processes. ACM Sigplan Notices **12**(8), 55–59 (1977)
6. Banken, H., Meijer, E., Gousios, G.: Debugging data flows in reactive programs. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). pp. 752–763. IEEE (2018)
7. Benveniste, A., Le Guernic, P.: Hybrid dynamical systems theory and the SIGNAL language. IEEE Tr. on Automatic Control **35**(5), 525–546 (1990)
8. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19**(2), 87–152 (1992)
9. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.A.: Static scheduling of multi-rate and cyclo-static DSP applications. In: Workshop on VLSI Signal Processing. IEEE Press (1994)
10. Bonér, J., Farley, D., Kuhn, R., Thompson, M.: The reactive manifesto (2014), `http://www.reactivemanifesto.org/`
11. Bonér, J., Klang, V., Kuhn, R., et al.: Akka library (2011-2019), `http://akka.io`
12. Brooks, C., Jerad, C., Kim, H., Lee, E.A., Lohstroh, M., Nouvellet, V., Osyk, B., Weber, M.: A component architecture for the internet of things. Proceedings of the IEEE **106**(9), 1527–1542 (September 2018)
13. Carreton, A.L., Mostinckx, S., Van Cutsem, T., De Meuter, W.: Loosely-coupled distributed reactive programming in mobile ad hoc networks. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. pp. 41–60. Springer (2010)
14. Chapiro, D.M.: Globally-Asynchronous Locally-Synchronous Systems. Ph.D. thesis, Stanford University (Oct 1984)
15. Cooper, G.H., Krishnamurthi, S.: Embedding dynamic dataflow in a call-by-value language. In: European Symposium on Programming. pp. 294–308. Springer (2006)
16. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. In: OSDI (2012)
17. Cremona, F., Lohstroh, M., Broman, D., Lee, E.A., Masin, M., Tripakis, S.: Hybrid co-simulation: it's about time. Software and Systems Modeling (November 2017)
18. Czaplicki, E., Chong, S.N.: Asynchronous functional reactive programming for GUIs. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation-PLDI'13. ACM Press (2013)
19. Dennis, J.B.: First version data flow procedure language. Report MAC TM61, MIT Laboratory for Computer Science (1974)
20. Edwards, J.: Coherent reaction. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. pp. 925–932. ACM (2009)

21. Eidson, J., Lee, E.A., Matic, S., Seshia, S.A., Zou, J.: Distributed real-time software for cyber-physical systems. Proceedings of the IEEE (special issue on CPS) **100**(1), 45–59 (2012)
22. Elliott, C., Hudak, P.: Functional reactive animation. In: ACM SIGPLAN Notices. vol. 32, pp. 263–273. ACM (1997)
23. Friedman, D.P., Wise, D.S.: The impact of applicative programming on multiprocessing. Indiana University, Computer Science Department (1976)
24. Gajski, D.: SpecC: Specification Language and Methodology. Kluwer Academic Publishers, Norwell, MA (2000)
25. Group, S.C.S.W., et al.: 1666-2011-IEEE standard for standard SystemC language reference manual
26. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE **79**(9), 1305–1319 (1991)
27. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science **410**(2-3), 202–220 (2009)
28. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. pp. 477–498. Springer Berlin Heidelberg, Berlin, Heidelberg (1985)
29. Harel, D.: Statecharts: A visual formalism for complex systems. Science of computer programming **8**(3), 231–274 (1987)
30. Hewitt, C.: Viewing control structures as patterns of passing messages. Journal of Artificial Intelligence **8**(3), 323–363 (1977)
31. Hewitt, C., Bishop, P.B., 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. pp. 235–245 (1973)
32. Kahn, G.: The semantics of a simple language for parallel programming. In: Proc. of the IFIP Congress 74. pp. 471–475. North-Holland Publishing Co. (1974)
33. Koopman, P.: A case study of toyota unintended acceleration and software safety (2014), `http://betterembsw.blogspot.com/2014/09/a-case-study-of-toyota-unintended.html`
34. Kuhn, R., Hanafee, B., Allen, J.: Reactive design patterns. Manning Publications Company (2017)
35. Lee, E., Reineke, J., Zimmer, M.: Abstract PRET machines. In: 2017 IEEE Real-Time Systems Symposium (RTSS). pp. 1–11 (Dec 2017)
36. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987)
37. Lee, E.A., Parks, T.M.: Dataflow process networks. Proceedings of the IEEE **83**(5), 773–801 (1995)
38. Lee, E.A.: The problem with threads. Computer **39**(5), 33–42 (2006)
39. Lee, E.A., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Transactions on Computer-Aided Design of Circuits and Systems **17**(12), 1217–1229 (1998)
40. Lee, E.A., Zheng, H.: Operational semantics of hybrid systems. In: Morari, M., Thiele, L. (eds.) Hybrid Systems: Computation and Control (HSCC). vol. LNCS 3414, pp. 25–53. Springer-Verlag (2005)
41. Lee, E.A., Zheng, H.: Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In: EMSOFT. pp. 114 – 123. ACM (2007)

42. Liao, S., Tjiang, S., Gupta, R.: An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In: Design Automation Conference. ACM (1997)
43. Liu, X., Matsikoudis, E., Lee, E.A.: Modeling timed concurrent systems. In: CONCUR 2006 - Concurrency Theory. vol. LNCS 4137, pp. 1–15. Springer (2006)
44. Lohstroh, M., Lee, E.A.: Deterministic actors. In: 2019 Forum for Specification and Design Languages (FDL). pp. 1–8 (Sep 2-4 2019)
45. Lohstroh, M., Schoeberl, M., Jan, M., Wang, E., Lee, E.A.: Work-in-progress: Programs with ironclad timing guarantees. In: 2019 International Conference on Embedded Software (EMSOFT) (Oct 2019)
46. Lohstroh, M., Schoeberl, M., Goens, A., Wasicek, A., Gill, C., Sirjani, M., Lee, E.A.: 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. pp. 152:1–152:4. ACM (2019)
47. Maler, O., Manna, Z., Pnueli, A.: From timed to hybrid systems. In: Real-Time: Theory and Practice, REX Workshop. pp. 447–484. Springer-Verlag (1992), uses super dense time (super-dense, superdense).
48. Matsikoudis, E., Lee, E.A.: The fixed-point theory of strictly causal functions. Technical Report UCB/EECS-2013-122, EECS Department, University of California, Berkeley (June 9 2013)
49. Meijer, E.: Reactive extensions (rx): Curing your asynchronous programming blues. In: ACM SIGPLAN Commercial Users of Functional Programming. pp. 11:1–11:1. CUFP '10, ACM, New York, NY, USA (2010)
50. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M.I., Stoica, I.: Ray: A distributed framework for emerging AI applications. simarXiv:1712.05889v2 [cs.DC] 30 Sep 2018 (2018)
51. Myter, F., Scholliers, C., De Meuter, W.: Distributed reactive programming for reactive distributed systems. arXiv preprint arXiv:1902.00524 (2019)
52. NASA Engineering and Safety Center: National highway traffic safety administration toyota unintended acceleration investigation. Technical assessment report, NASA (January 18 2011)
53. Parks, T.M.: Bounded scheduling of process networks. Ph.D. Thesis Tech. Report UCB/ERL M95/105, UC Berkeley (1995)
54. Prokopec, A.: Pluggable scheduling for the reactor programming model. In: Ricci, A., Haller, P. (eds.) Programming with Actors: State-of-the-Art and Research Perspectives, pp. 125–154. Springer International Publishing (2018)
55. Roestenburg, R., Bakker, R., Williams, R.: Akka In Action. Manning Publications Co. (2016)
56. Schoeberl, M., Puffitsch, W., Hepp, S., Huber, B., Prokesch, D.: Patmos: A time-predictable microprocessor. Real-Time Systems **54(2)**, 389–423 (Apr 2018)
57. Tasharofi, S., Dinges, P., Johnson, R.E.: Why do scala developers mix the actor model with other concurrency models? In: European Conference on Object-Oriented Programming. pp. 302–326. Springer (2013)
58. Tilkov, S., Vinoski, S.: Node. js: Using javascript to build high-performance network programs. IEEE Internet Computing **14**(6), 80–83 (2010)
59. Torres Lopez, C., Gurdeep Singh, R., Marr, S., Gonzalez Boix, E., Scholliers, C.: Multiverse debugging: Non-deterministic debugging for non-deterministic programs (2019)
60. Vernon, V.: Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley Professional (2015)

61. Zhao, Y., Lee, E.A., Liu, J.: A programming model for time-synchronized distributed real-time systems. In: Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 259 – 268. IEEE (2007)
62. Zhao, Y., Liu, J., Lee, E.A.: A programming model for time-synchronized distributed real-time systems. In: 13th IEEE Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07. pp. 259 – 268 (April 2007)
63. Zimmer, M., Broman, D., Shaver, C., Lee, E.A.: FlexPRET: A processor platform for mixed-criticality systems. In: Real-Time and Embedded Technology and Application Symposium (RTAS) (2014)

A Summary of the Reactor model

Execution environment objects	
Set of identifiers	Σ (an abstract set)
Set of values	$\mathfrak{V}$ (an abstract set)
Absent value	$\varepsilon \in \mathfrak{V}$
Set of priorities	$\mathcal{P} = \mathbb{Z}^- \cup \mathbb{Z}^+ \cup \{*\}$
Event queue	$\mathcal{Q}_E$
Reaction queue	$\mathcal{Q}_R$
Logical time	t
Physical time	T
Set of tags	$\mathbb{T} = \mathbb{N}^2$
Set of origins	$\mathfrak{O} = \{\text{Logical}, \text{Physical}\}$
Reactors	
Reactor instance	$r = (I, O, A, S, \mathcal{N}, \mathcal{M}, \mathcal{R}, \mathcal{G}, P, \bullet, \diamond)$
Set of input ports for r	$I(r) \subseteq \Sigma$
Set of output ports for r	$O(r) \subseteq \Sigma$
Set of actions for r	$A(r) \subseteq \Sigma \times \mathbb{N} \times \mathfrak{O}$
Initialization action for r	$\bullet(r) \in A(r)$
Termination action for r	$\diamond(r) \in A(r)$
Set of state identifiers for r	$S(r) \subseteq \Sigma$
Set of reactions contained in r	$\mathcal{N}(r)$
Set of mutations contained in r	$\mathcal{M}(r)$
Set of contained reactors of r	$\mathcal{R}(r)$
Topology of reactors in $\mathcal{R}(r)$	$\mathcal{G}(r) \subseteq \left(\bigcup_{r' \in \mathcal{R}(r)} O(r')\right) \times \left(\bigcup_{r' \in \mathcal{R}(r)} I(r')\right)$
Priority function	$P : \mathcal{N} \cup \mathcal{M} \to \mathcal{P}$
Reactor containing reactor r	$C(r)$
Inputs and outputs	
Input, output instance	$i, o \in \Sigma$
Reactions dependent on $i \in I(r)$	$\mathcal{N}(i) = \{n \in \mathcal{N}(C(i)) \mid i \in D(n)\}$
Reactions antidependent on $o \in O(r)$	$\mathcal{N}(o) = \{n \in \mathcal{N}(C(o)) \mid o \in D^\vee(n)\}$
Actions	

Action instance	$a = (x, d, \mathfrak{o})$
Action identifier	$x \in \Sigma$
Action delay	$d \in \mathbb{T}$
Action origin	$\mathfrak{o} \in \mathfrak{O}$
Events	
Event instance	$e = (\mathfrak{t}, \mathfrak{v}, \mathfrak{g})$
Event trigger	$\mathfrak{t} \in \Sigma$
Event value	$\mathfrak{v} \in \mathfrak{V}$
Event tag	$\mathfrak{g} \in \mathbb{T}$
Set of reactions and mutations triggered by trigger $\mathfrak{t}$	$\mathcal{T}(\mathfrak{t}) = \{k \in \mathcal{N}(C(\mathfrak{t})) \cup \mathcal{M}(C(\mathfrak{t})) \mid \mathfrak{t} \in \mathcal{T}(k)\}$
Reactions	
Reaction instance	$n = (D, \mathcal{T}, B, D^{\vee}, H)$
Set of reaction dependencies	$D(n) \subseteq I(C(n)) \cup \left(\bigcup_{r \in \mathcal{R}(C(n))} O(r)\right)$
Set of reaction triggers	$\mathcal{T}(n) \subseteq D(n) \cup x(A(C(n)))$
Reaction body	$B(n)$
Set of reaction antidependencies	$D^{\vee}(n) \subseteq O(C(n)) \cup \left(\bigcup_{r \in \mathcal{R}(C(n))} I(r)\right)$
Set of schedulable actions	$H(n) \subseteq x(A(C(n)))$
Reactor containing reaction n	$C(n)$
Reaction priority	$P(n) \in \mathbb{Z}^+ \cup \{*\}$
Priority of unordered reactions	$\forall q \in \mathbb{Z}^- \, \forall p \in \mathbb{Z}^+. \; (n < *) \wedge (p \not> *) \wedge (* \not> p) \wedge (* \leq *)$
Mutations	
Mutation instance	$m = (D, \mathcal{T}, B, D^{\vee}, H)$
Set of mutation dependencies	$D(m) \subseteq I(C(m))$
Set of mutation triggers	$\mathcal{T}(m) \subseteq D(m) \cup x(A(C(m)))$
Mutation body	$B(m)$
Set of mutation antidependencies	$D^{\vee}(m) \subseteq O(C(m)) \cup \left(\bigcup_{r \in \mathcal{R}(C(m))} I(r)\right)$
Set of schedulable actions	$H \subseteq x(A(C(m))) \cup \{x(a) \mid \forall r \in \mathcal{R}(C(x)).\ a \in \{\bullet(r), \diamond(r)\}\}$
Reactor containing mutation m	$C(m)$
Mutation priority	$P(m) \in \mathbb{Z}^-$