Deterministic Actors

Marten Lohstroh, Edward A. Lee
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA, USA
{marten,eal}@berkeley.edu

***Abstract*—Actors have become widespread in programming languages and programming frameworks focused on parallel and distributed computing. While actors provide a more disciplined model for concurrency than threads, their interactions, if not constrained, admit nondeterminism. As a consequence, actor programs may exhibit unintended behaviors and are less amenable to rigorous testing. We show that nondeterminism can be handled in a number of ways, surveying dataflow dialects, process networks, synchronous-reactive models, and discrete-event models. These existing approaches, however, tend to require centralized control, pose challenges to modular system design, or introduce a single point of failure. We describe "reactors," a new coordination model that combines ideas from several of the aforementioned approaches to enable determinism while preserving much of the style of actors. Reactors promote modularity and allow for distributed execution. By using a logical model of time that can be associated with physical time, reactors also admit control over timing.**

***Index Terms*—concurrency control, distributed computing, programming, software testing**

I. Introduction

Loosely, actors are concurrent objects that communicate by sending each other messages. Under this loose definition, an enormous number of actor programming languages and models have been developed, although many are called by other names, including dataflow, process networks, synchronous-reactive languages, and discrete-event languages. A narrower definition, originally developed by Hewitt and Agha [1], [2], has recently seen a renaissance, appearing in several software frameworks such as Scala actors [3], Akka [4], and Ray [5], and programming languages, like SALSA [6] and Rebeca [7]. Unlike various related dataflow models, the Hewitt actor model, as it is known, is nondeterministic. Even if each actor reacts in deterministic ways to incoming messages, when composed, a collection of actors typically yields many possible behaviors. In this paper, we explain how most of the nondeterminism can be avoided without losing expressivity. This results in distributed programs with more testable and repeatable behaviors that admit nondeterminism only where it is explicitly required by the application.

We begin by illustrating the concern with a simple example, given in Fig. 1. It uses a pseudo-code syntax that is a mashup of several of the concrete languages mentioned above. This

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.

```
1  actor X {
2    state = 1;
3    handler dbl(){
4      state *= 2;
5    }
6    handler incr(arg){
7      state += arg;
8      print state;
9    }
10 }

11 actor Y {
12   handler main {
13     x = new X();
14     x.dbl();
15     x.incr(1);
16   }
17 }
```

Fig. 1. Pseudo code for an actor network that is deterministic under reasonable assumptions about message passing.

code defines an actor class `X` that has a single integer state variable that is initialized to 1. It has two message handlers, named `dbl()` and `incr()`. When invoked, these handlers will double and increment the state variable, respectively.

The actor named `Y` with handler **main** creates an instance of `X` and sends it two messages, `dbl` and `incr`. Note that although many actor languages make these look like remote procedure calls, presumably because such syntax is familiar to programmers, they are not remote procedure calls. Lines 14 and 15 send messages and return immediately. The semantics of actors is "send and forget," a key feature that enables parallel and distributed execution.

The program in Fig. 1 is deterministic under mild assumptions about the network that relays messages. First, we need to assume that messages are delivered reliably in the same order that they are sent.[1] Since `dbl` is sent before `incr`, actor `x` will execute handler `dbl()` before handler `incr()`. Second, we need to assume that handlers are mutually exclusive.[2] That is, once a handler begins executing, it executes to completion before any other handler in the same actor begins executing. This assumption prevents a race condition between lines 4 and 7. Thus, in this program, line 4 will execute before line 7 and the printed output will be 3.

Consider now the seemingly minor elaboration shown in Fig. 2. This program introduces a third actor class, `Relay`, which has a single handler `rly` that simply relays a message, in this case `dbl`, to the actor `x` passed to it. This is about as close as one can get to a "no op" in actor-oriented program. It

[1]This can be realized on a distributed system by relying on the eventual, in-order delivery property of TCP/IP.

[2]This assumption can be relaxed by statically analyzing the code of the handlers and enforcing mutual exclusion only between handlers that share state variables.

```
actor Y {
  handler main {
    x = new X();
    z = new Relay();
    z.rly(x);
    x.incr(1);
  }
}

actor Relay {
  handler rly (x){
    x.dbl();
  }
}
```

Fig. 2. Modification of the code in Fig. 1 yielding a nondeterministic program. Actor X remains the same.

```
import ray
@ray.remote
class X():
  def __init__(self):
    self.count = 1
  def dbl(self):
    self.count *= 2
    return self.count
  def inc(self, arg):
    self.count += arg
    return self.count
@ray.remote
class Relay():
  def rly(self, x):
    return ray.get(
      x.dbl.remote())

def test():
  x = X.remote()
  r = Relay.remote()
  f1 = r.rly.remote(x)
  f2 = x.inc.remote(1)
  return ray.get(f1) \
    + ray.get(f2)

ray.init()
result = test()
print(result)
```

Fig. 3. A nondeterministic actor network in the syntax of Ray.

is an actor that, when it receives a message, simply passes the message on. However, this innocent change has profound consequences. The execution is no longer deterministic under any reasonable assumptions about message delivery. The printed value could be either 2 or 3, depending on whether `dbl()` or `incr()` is invoked first. (The final state will be 3 or 4.)

A similar example written in the concrete syntax of Ray [5] is shown in Fig. 3. Ray extends the metaphor of remote procedure calls by integrating futures [8] into the language. In Ray, message handlers can return values. The semantics is still "send and forget," so when a message is sent, a "future" is returned. A future is a placeholder data structure for the returned result. Execution can continue until returned result is actually needed, at which point the sender of the message can call `ray.get()` on the future. The call to `ray.get()` blocks until the result is actually received. Nevertheless, the program in Fig. 3 remains nondeterministic; it is capable of producing either 5 or 6 as a result. You can easily verify this by inserting `sleep()` statements from Python's time module to alter the timing of the execution.

The blocking behavior of `ray.get()` provides a mechanism, one not available in any other actor language that we know of, for controlling the execution of a network of actors.

```
def test():
  x = X.remote()
  r = Relay.remote()
  f1 = r.rly.remote(x)
  part = ray.get(f1)
  f2 = x.inc.remote(1)
  return part \
    + ray.get(f2)
```

Fig. 4. Modification to the program in Fig. 3 to make it deterministic.

This mechanism could be used, for example, to make the program in Fig. 3 deterministic. The `test` function could be replaced with the code in Fig. 4. This code forces the main actor to block until the result of the invocation of `dbl()` is received before sending the `incr` message. This solution, however, requires a very savvy programmer and largely defeats the purpose of the futures. We doubt that many Ray programs will be written with such controls.

This type of nondeterminism is endemic to the Hewitt actor model. Moreover, without the blocking futures of Ray, it is difficult to change the program in Fig. 2 to consistently print 3. One way would be to modify class `X` so that it *always* invokes `dbl()` before `incr()`, but this is a much more restrictive actor that may as well have only one message handler that doubles the state and then increments it. Alternatively, we could set up another message handler in `X` that tells it which handler to invoke first, but we would have to ensure that messages to that handler are invoked before any other. Moreover, the semantics now becomes complex. Should a message telling `X` to invoke `dbl()` first apply only to the next `dbl` message or to all subsequent ones? What if two `dbl` messages arrive with no intervening `incr` message?

Since such a simple program results in unfixable nondeterminism, we can only conclude that the Hewitt actor model should be used only in applications where determinism is not required. While there are many such applications, even for those, we pay a price. The code becomes much more difficult to test. Standard testing techniques are based on presenting input test vectors and checking the behavior of the program against results known to be good; in the face of non-determinism, the entire set of known-good results may be difficult to determine and too vast to enumerate.

To underscore the challenges that nondeterministic software poses to testability, we cite Toyota's unintended acceleration case. 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 NASA study [9] was unable to find a "smoking gun," but they concluded that the software was "untestable" and that it was impossible to rule out the possibility of unintended acceleration [10]. The software used a style of design that tolerates a seemingly innocuous form of nondeterminism. Specifically, many state variables, representing for example the most recent readings from a sensor, were accessed unguarded by a multiplicity of threads. We suspect that this style of design seemed reasonable to the software engineers because one should always use the "most recent" value of a sensor. But the software becomes untestable because, given any fixed set of inputs, the number of possible behaviors is vast.

Not all concurrent software is used in such safety-critical scenarios, of course, but all software benefits from testability. The Toyota software did not use Hewitt actors, but many Hewitt actor programs share a similar form of nondeterminism. Messages are handled in order of arrival, so the state of an

actor represents the effects of the "most recent" messages.

The main contribution of this paper is to show that the Hewitt actor model can be extended to yield a deterministic model of computation using any of various techniques, some of which have a long history. These include various dataflow dialects, process networks, synchronous-reactive models, and discrete-event models. After discussing these coordination approaches, we explain a new coordination model we call "reactors," which combines several of the aforementioned techniques with the goal of enabling determinism while preserving much of the style of actors. Specifically, the reactor model promotes modularity and allows for distributed execution. Reactors use a logical model of time to achieve deterministic execution; we show how their capability of relating logical time to physical time allows for the design of distributed reactor systems that behave deterministically.

II. Achieving Determinism

A system is deterministic if, given an initial state and a set of inputs, it has exactly one possible behavior. For this definition to be useful, we have to define "state," "inputs," and "behavior." For example, if we include in our notion of "behavior" the timing of actions, then no computer program in any modern programming language is deterministic. In our discussion above, the actor programs have no inputs, the initial state is `state = 1` in an instance of actor `X`, and the "behavior" is the result printed. Timing is not part of the model and therefore irrelevant to the definition of determinism.

Determinism is a property of a model, not a property of a physical realization of a system [11]. A Turing machine, for example, provides a deterministic model of computation that does not include timing. The "input" is a sequence of bits, and the "behavior" consists of sequential transformations of that sequence. Any particular physical realization of a Turing machine will have properties that are absent from Turing's model, such as timing, but we could construct a different model that did consider timing part of the "behavior." Such a model would be nondeterministic. The same physical system, therefore, is deterministic or not depending on the model.

A. Determinism for Software

Whether a software system is deterministic depends on our model of the software. A simple model of a program defines initial state as the starting values of all variables, the inputs as a static bit sequence (a binary number) available all at once at the start of execution, and the output as a bit sequence produced all at once upon termination of the program. This is the classic Church-Turing view of computation.

This classic model, however, has difficulty with many practical software systems. A web server, for example, does not have inputs that can be defined as a binary number available all at once at the start of execution. Nor does it terminate and produce a final output. An alternative model for a web server defines its inputs as a (potentially unbounded) *sequence* of binary numbers, and the "behavior" as sequence of binary numbers produced as outputs. In this model, whether the web server is deterministic may be an important question.

In a concurrent or distributed software system, however, defining the inputs as a *sequence* of binary numbers may be problematic. A distributed database, like Google Spanner [12], for example, accepts inputs at a globally distributed collection of data centers. It is impossible to tell whether a query arriving in Dallas arrives before or after a query arriving Seattle.[3] In Google Spanner, however, when a query comes in to a data center, it is assigned a numerical timestamp. The "inputs" to the global database are defined as an unbounded collection of timestamped queries, and the "behavior" is the set of responses to those queries. Under this model, Spanner is deterministic. We emphasize that this is not an assertion about any *physical* realization of Spanner, which could exhibit behaviors that deviate from the model (if, for example, hardware failures undermine the assumptions of the model). It is the *model* that is deterministic, not the physical realization.

Consider again the actor programs in Figs. 2 and 3. If we wish for these programs to be deterministic, we have to somehow constrain the order in which message handlers are invoked. We have an intuitive expectation that `dbl()` should be invoked before `incr()`, *but that is not what the programs say*. The programs, as written and as interpreted by modern actor frameworks, do not specify the order in which these handlers should be invoked. Thus, it will not be sufficient to simply improve the implementation of the actor framework. We have to also change the model.

B. Coordination for Determinism

Let us focus on the actor network sketched in Fig. 2. Since actor `Y` first sends a message that has the eventual effect of doubling the state of actor `X` and then sends a second message to increment the state of `X`, let us assume that it is the design intent that the doubling occur before the incrementing. Any technique that ensures this ordering across a distributed implementation will require some coordination. There are many ways to accomplish this, many of which date back several decades. Here, we will outline a few of them.

In 1974, Gilles Kahn showed that networks of asynchronously executing processes could achieve deterministic computation and provided a mathematical model for such processes (Scott-continuous functions over sequence domains) [13]. In 1977, Kahn and MacQueen showed that a very simple execution policy using blocking reads guarantees such determinacy [14]. Using the Kahn-MacQueen principle, actor `X` in Fig. 2 could be replaced with `X_KPN` (for Kahn Process Network) in Fig. 5. Instead of separate message handlers, a process in a KPN is a single threaded program that performs blocking reads on inputs. The `await` calls in Fig. 5 perform such blocking reads. That code ensures that doubling the state will occur before incrementing it even if actor `Y` sends its output messages in opposite order.

[3] Fundamentally, it is not only difficult to decide which query arrives first, it is impossible to even define what this means. Under the theory of relativity, the ordering of geographically separated events depends on the observer.

```
actor X_KPN  {
  handler main {
    state = 1;
    await(dbl);
    state *= 2;
    arg = await(incr);
    state += arg;
    print state;
  }
}
```

Fig. 5. Variant of X in Fig. 2 to encode design intent using blocking reads.

```
actor Relay {
  handler rly (X x) {
    if (some condition) { x.dbl(); }
  }
}
```

Fig. 6. Modification to actor Relay in Fig. 2 to filter messages.

This way of encoding the design intent, however, has some disadvantages. Suppose that the Relay actor, instead of just relaying messages, filters them according to some condition, as shown in Fig. 6. Now the X_KPN will permanently block awaiting a dbl message. The filtering logic would have to repeated in the X_KPN actor, which would have to surround the blocking read of dbl with a conditional. Moreover, the condition would have to be available now to X_KPN, making the Relay actor rather superfluous. Indeed, our experience building KPN models is that conditionals tend to have to be replicated throughout a network of connected actors, thereby compromising the modularity of the design.

Another family of techniques that are used to coordinate concurrent executions for determinism fall under the heading of dataflow and also date back to the 1970s [15], [16]. Modern versions use carefully crafted notions of "firing rules" [17], which specify preconditions for an actor to react to inputs. Actors can dynamically switch between firing rules governed by some conditions, but once again the conditions need to be shared across a distributed model to maintain coordination. One particularly elegant mechanism for governing such sharing is scenario-aware dataflow, where a state machine governs the coordinated switching between firing rules [18].

Another family of coordination techniques that can deliver deterministic execution uses the synchronous-reactive (SR) principle [19]. Under this principle, actors (conceptually) react simultaneously and instantaneously at each tick of a global (conceptual) clock. Like Kahn networks, the underlying semantics is based on fixed points of monotonic functions on a complete partial order [20] and determinism is assured. Unlike Kahn networks, however, the global clock provides a form of temporal semantics. This proves valuable when designing systems where time is important to the behavior of the system, as is the case with many cyber-physical systems. Some generalizations include multiclock versions [21]. Many projects have demonstrated that despite the semantic model of simultaneous and instantaneous execution, it is possible to implement such models in parallel and on distributed machines using strategies generally called physically asynchronous, logically synchronous (PALS) [22].

A fourth alternative, and the one we focus on in Sec. III, is based on discrete-event (DE) systems, which have historically been used for simulation [23], [24], but can also be used as a deterministic execution model for actors. DE is a generalization of SR, where there is a quantitative measure of time elapsing between ticks of the global clock [25]. In DE models, every message sent between actors has a timestamp, which is a numerical value, and all messages are processed in timestamp order. The underlying semantics of these models is based on generalized metric spaces rather than complete partial orders, but this semantics similarly guarantees determinism [26].

One challenge when reasoning about DE systems is that we immediately face (at least) two time lines: logical time and physical time(s). Logical time is the time of timestamps. Physical time is the time of physical clocks (not to be confused with the semantic clocks of synchronous-reactive models), and in a distributed system, there are typically multiple physical clocks. In real-time distributed systems, logical time and physical time can be aligned at selected points (e.g., when taking data from sensors or when actuating some physical devices), but otherwise, logical time is used as a semantic mechanism to ensure deterministic ordering and is allowed to differ from physical time. Unfortunately, natural language makes it difficult to form sentences about more than one time line together. Careful wording is required to avoid confusion, and sometimes, no wording seems adequate.

III. Reactors

Reactors, introduced in [27], are deterministic actors composed out of *reactions* and coordinated under a DE semantics. Reactions are message handlers, and messages between reactors are timestamped and handled in timestamp order. Messages with identical timestamps are logically simultaneous and are handled in a deterministic order. We use the term "reactors" to distinguish them from Hewitt actors. We are currently developing a meta-language called Lingua Franca (LF) for defining reactors and their compositions. LF is a polyglot meta language in which the logic of reactors is given in some target language such as C, Java, or JavaScript. The program from Fig. 2 is rewritten in LF in Fig. 7 with the target language C, where all text between the delimiters {= ... =} is C code.

A. Ports

Reactors do not directly refer to their peers. Instead, they have named (and typed) input and output ports. Timestamped incoming messages arrive at input ports, and reactions send timestamped messages via output ports.

B. Hierarchy

A composite reactor contains interconnected instances of reactors. The composite named "Main" in Fig. 7 contains one instance of each of the three other reactors and defines how their ports are connected. In this simple example, there is only one composite, but in LF, composites themselves can have input and output ports and can contain composites.

```
1  reactor X {
2    input dbl;
3    input incr:int;
4    preamble {=
5      int state = 1;
6    =}
7    reaction(dbl) {=
8      state *= 2;
9    =}
10   reaction(incr) {=
11     state += incr;
12     print state;
13   =}
14 }
15 reactor Relay {
16   input r;
17   output out;
18   reaction(r)->out {=
19     set(out, r);
20   =}
21 }
22 reactor Y {
23   output dbl:void;
24   output incr;
25   timer t(0);
26   reaction(t) {=
27     set(dbl, void);
28     set(incr, 1);
29   =}
30 }
31 composite Main {
32   x = new X();
33   r = new Relay();
34   y = new Y();
35   y.dbl -> r.r;
36   r.out -> x.dbl;
37   y.incr -> x.incr;
38 }
```

Fig. 7. Lingua Franca code for a reactor network.

C. Logical Time

Messages in LF have timestamps. Reactions to messages occur at a logical time equal to the timestamp, and logical time does not advance during the reaction. An input port can have at most one message at any logical time. If no message is present, then the input is *absent* at that time, and no message can later appear with that timestamp. Any output messages produced in a reaction bears the same timestamp as the input message that triggers the reaction. These outputs, therefore, are logically simultaneous with the triggering inputs. If any reactor receives more than one message with the same timestamp (on distinct input ports), those messages are logically simultaneous. If they trigger multiple reactions, those reactions will be deterministically invoked in the order that the reactions are defined. A reactor may also have one or more *timers*, as shown on line 25 of Fig. 7. That timer, named `t`, triggers once with delay 0 (at the logical start time of execution), causing the reaction to `t` to be invoked at the logical start time. That reaction sends two messages, both timestamped with the same timestamp. Reactions to those messages will be invoked at the same logical time. Timers support periodic and one-time actions.

D. Scheduling

Reactions may share state with other reactions in the same reactor. To preserve determinacy, reactions within one reactor are invoked in a predefined order when there are logically simultaneous input messages. Semantically, this approach follows the sequential constructiveness principle of SCCharts [28], which allows arbitrary sequential reads or writes of shared variables during a synchronous-reactive tick.

If two distinct reactors receive logically simultaneous messages, then their reactions may be invoked in parallel *unless there is a direct dependency between them.* In Fig. 7, the `Relay` reactor's reaction to input `r` declares that it writes to output `out` using the syntax on line 18. This means that there is a direct feedthrough relationship from input `r` to output `out`. At the level of the composite, this information can be used to analyze dependencies within the composite and impose scheduling constraints (an algebra for such analysis is given in [29]). The purpose of these scheduling constraints is to ensure the deterministic DE semantics even in the presence of parallel and distributed execution.

One of the goals of LF is to abstract code written in the target language. The code in between {= ... =} delimiters is not even parsed, much less analyzed. In principle, it is possible to infer the declared input/output dependencies from that code if it were to be parsed. However, if the reading of input messages or writing of output messages in a reaction is data dependent, then whether a declared dependency is actually a real dependency proves undecidable. Hence, even the most sophisticated analysis will be conservative.

While declaring the dependencies statically comes at the cost of a slight loss in the accuracy of the reporting of causal dependencies, it facilitates the *polyglot* nature of Lingua Franca. A variety of target languages can be supported by the same model. For example, using C as a target language is appropriate for resource constrained, deeply embedded systems, while Python may be a better choice for AI applications and Java for enterprise-scale distributed applications. Because target-language code is not analyzed in the LF compiler, comparatively little effort is required to add support for new target languages.

IV. Distributed Execution

Discrete-event models of computation, where time-stamped events are processed in timestamp order, have been used for simulation for a long time [23], [24]. There is also a long history of executing such simulations on parallel and distributed platforms, where the primary challenge is maintaining the timestamp ordering without a centralized event queue. The classic Chandy and Misra approach [30] assumes reliable eventual in-order delivery of messages and requires that before any actor with two or more input ports process any timestamped input message, that every input have at least one pending input message. It is then safe to process the message with the least timestamp. To avoid starvation, the Chandy and Misra approach requires that null messages be sent periodically on every channel so that no actor is blocked indefinitely waiting for messages that will never arrive.

The Chandy and Misra approach is the centerpiece of a family of so-called "conservative" distributed simulation techniques. An alternative, first described by Jefferson [31], is to use speculative execution. Jefferson's so-called "time warp" approach relies on checkpointing the state of all actors and the event queue and then handling time-stamped messages as they become available. As messages are handled, the local notion of "current time" is updated to match the timestamp of the message. If a message later becomes available that has a timestamp earlier than current time, then the simulation is rolled back to a suitable checkpoint and redone from that point.

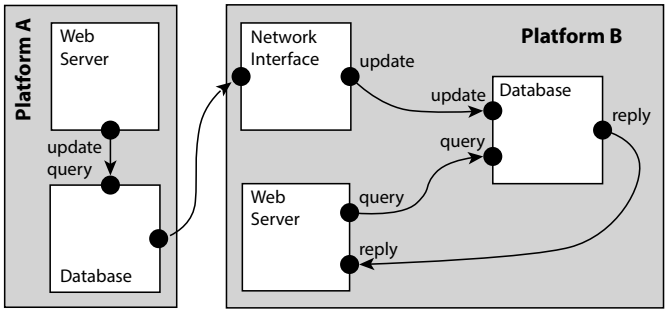

Fig. 8. A distributed system using reactors with Ptides.

```
1 reactor WebServer() {
2   output out:string;
3   timer t(0);
4   action incoming:string;
5   reaction(t) -> incoming {=
6     ... set up server ...
7     startServer(function(query) {
8       schedule(incoming, query);
9     });
10  =}
11  reaction(incoming) -> out {=
12    set(out, incoming);
13  =}
14 }
```

Fig. 9. Sketch of a reactor for Web Server in Fig. 8.

While both of these techniques are effective for simulation, they have serious disadvantages for reactors, which are intended to be used as system implementations, not as simulations. In addition to the overhead of null messages, the Chandy and Misra approach suffers the more serious disadvantage that every node in a distributed system becomes a single point of failure. If any node stops sending messages, all other nodes will eventually grind to a halt, unable to proceed while they wait for null or real messages. In addition to the overhead of redoing execution, the time warp approach suffers the more serious disadvantage that in a system deployment, unlike a simulation, some actions cannot be rolled back.

To address these concerns, Zhao et al. introduced Ptides, a programming model for distributed real-time system realizations (not simulations) that is compatible with reactors [32]. Ptides was later independently reinvented at Google, where it became the backbone of a globally distributed database system called Spanner [12].

Ptides and Spanner make two key assumptions about the execution platform. First, they assume that each node in the distributed system has a physical clock that is synchronized with that of all other nodes, and that there is a bound E on the clock synchronization error. Second, they assume that every network connection between nodes has a bound L on the latency for message delivery.

We can use Spanner's database application in a simple scenario depicted in Fig. 8 to explain how these two assumptions enable efficient and deterministic distributed execution. Consider a distributed database where the data is replicated on two different platforms, A and B in the figure. Assume that the two copies of the database are initially identical and that an update query arrives through Web Server on Platform A that makes a change to a record in the database.

Web Server can be realized as a reactor like that sketched in Fig. 9. At the logical start time of the execution, the first reaction sets up the server to listen for incoming messages, and then starts the server, providing a callback function to invoke when there is an incoming query. When an incoming query arrives, the `schedule` function is invoked to request that the action named `incoming` be triggered at the next available logical time. As explained in [27], this action will timestamp the action using its local physical clock and trigger the second reaction at a logical time equal to that timestamp. The second reaction will forward the timestamped message to the `Database` reactor. The `Database` reactor will broadcast to all other replicas of the database the update to the record, as indicated by the dashed line. That broadcast incurs network latency that is assumed to not exceed some number L.

At around that same time that Platform A receives the update query, suppose that Platform B receives a query for the value of the same record being updated at Platform A. How should the system respond? In Spanner (and Ptides), this query at Platform B will also be timestamped using the local physical clock, and the semantics of the system defines the correct response to depend on the numerical order of the timestamps of the two queries. If the query at Platform A has an earlier or equal timestamp to that at Platform B, then the correct response is the updated record value. Otherwise, the correct response is the value before the update.

Suppose that Platform B, at the `Database` reactor, has an input message with timestamp t. Can it safely handle that message? To be safe, it has to be sure that it will not later receive a message on its update port with a timestamp earlier than or equal to t. How can it be sure?

Such a distributed system could use the Chandy and Misra approach, which would require Platform A to periodically send timestamped null messages to Platform B. Then, at Platform B, the `Database` reactor will repeatedly receive null messages on its `update` port with steadily increasing timestamps. As soon as one of those timestamps exceeds t, it can handle the message on its `query` port that has timestamp t and send a reply back to `WebServer`. However, as we have pointed out, the Chandy and Misra approach has high overhead and is vulnerable to node failures.

In Ptides and Spanner, the approach instead is to watch the local clock, and when it exceeds $t+E+L$, process the `query` message. If the clock synchronization error indeed does not exceed E, and the network latency indeed does not exceed L, then the `query` message is safe to process. No message with a timestamp earlier than or equal to t will later arrive.

The same strategy could be applied to correct the nondeterminism in Fig. 2. In that example, the messages are all logically simultaneous (they bear the same timestamp). If

the three actors are executed on distributed machines, then when the machine executing X receives a timestamped incr message with timestamp t, it should not invoke the message handler until the local clock exceeds $t+E+2L$. The $2L$ in this threshold is a consequence of structure of the model, where two network traversals are involved. The use of hierarchy ensures that there is a software entity, the container for the three actors, that "knows" the topology, and the use of ports with causality interfaces ensures that the dependency analysis required to derive this threshold can be performed.

The cost of determinism in this case is increased latency because the machine executing X must wait for physical time to pass. If this cost is too high, and nondeterminism is tolerable, then the program can be rewritten using the pattern in Fig. 9 to receive messages from the network. In that pattern, an incoming message is assigned a new timestamp upon being received, and no waiting is required to process the message. The principle we advocate is that the system designer should *choose* to make the system nondeterministic, rather than having this decision forced by the framework. Moreover, once a timestamp is assigned, the behavior of the system is deterministic. As a consequence, even a nondeterministic design becomes testable because input test vectors can include the assigned timestamps as part of the test vector.

V. Performance and Robustness

The above example assumes that execution times of reactions and network interfaces are negligible. In Spanner's database application, this may be a reasonable assumption. A system requirement might be, for example, that Platform B should respond to any query arriving at Web Server within 30ms. In this case, the system requirement is that $E + L$, plus any execution times involved, be less than 30ms. The execution times here are indeed likely to be negligible compared to 30ms, so they can be safely neglected.

In many embedded control applications, however, execution times will not be negligible. In such cases, the threshold for a message with timestamp t to be safe to process becomes $t + E + L + T$, where T is a bound on the execution time of all code in the path from Web Server at Platform A and the Database reactor at Platform B. At considerable cost in effort, and with many caveats, these execution times could be bounded using well-established worst-case-execution-time (WCET) analysis [33]. Or the system could be realized using PRET machines [34], in which case extremely high confidence in the bounds on the execution times becomes achievable.

The correctness of the Ptides/Spanner message handling depends on assumptions about the underlying execution platform. Of course, this is true of any engineered system, and it is always possible for assumptions to be violated in the field. One key advantage of the Pitdes/Spanner approach is that such violations are observable when they lead to violations of the discrete-event semantics. Suppose that at Platform B, the Database reactor has processed a message with timestamp t and issued a reply, and it then later receives an update message with timestamp $t' < t$. At this point, the runtime system at Platform B knows that one of the assumptions was violated. Either the real-time clocks have a discrepancy higher than E, the network latency exceeds L, or the execution times are not negligible. It is impossible to tell which of these assumptions was violated, but we can be sure that one of them was violated, and the system can declare a fault condition. At this point, the system has to switch to fault handling mode.

How to handle faults depends on the application. In Spanner, which is a distributed database application, a transaction schema is overlaid on the Ptides approach. When faults are detected, transactions are rejected, and the state is rolled back. In real-time control applications, rollback may not be possible, so faults may be much more costly. For such applications, the assumptions about E, L, and execution times need to be more conservative and fault handlers need to be more aggressive.

VI. Related Work

The use of synchronous-reactive principles to deterministically coordinate concurrent software has a long history, with notable contributions like Reactive C [35], SL [36], SyncCharts [37], and ReactiveML [38]. A modern variant of SyncCharts, SCCharts [39], composes finite state machines under a synchronous semantics. It has been recently augmented with a semantic notion of time [40] based on the concept of dynamic ticks [41]. Like reactors, components can inform the scheduler at what logical time to trigger reactions.

Also related are a whole family of so-called "active object" languages [42]. These languages approach the problem of concurrent execution by generalizing object-oriented programming with asynchronous method calls and (sometimes) futures, techniques that allow for parallel and distributed execution. Ensuring determinacy, however, is not a priority, and even support for avoiding the common pitfalls of threads [43] is sparse in some of these languages.

The concept of "reactive isolates" [44] (later also called "reactors") was introduced by Prokopec and Odersky to modularly combine different communication protocols inside the same actor. Their implementation is the Scala-based Reactors.IO framework [45]. A key difference with Hewitt actors is that reactive isolates have separate channels for receiving messages from other actors and internal event streams to compose reactions. Their channels are analogous to our input ports. They have no analogy to our output ports, however. A channel in reactive isolates is a direct reference to an isolate that other isolates can send messages to. Like classic actors, reactive isolates do not feature a semantic notion of time, and their communication is asynchronous with no guarantees on message arrival order.

VII. Conclusion

Reactors are a variation on actors that leverage logical time, following classical synchronous-reactive principles, to achieve determinism locally. The reactor model also relates logical time to *physical* time, which allows for the specification of real-time constraints, as well as the preservation of determinism across distributed systems via the application

of safe-to-process analysis known from Spanner and Ptides. While the emphasis of reactors is on determinacy, asynchrony and nondeterminism can be realized, when needed by an application, via the dynamic scheduling of sporadic events.

ACKNOWLEDGMENT

The authors thank Andrés Goens for comments on an earlier version of this paper.

REFERENCES

[1] C. Hewitt, "Viewing control structures as patterns of passing messages," *Journal of Artificial Intelligence*, vol. 8, no. 3, pp. 323–363, 1977.

[2] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, no. 1, pp. 1–72, 1997.

[3] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2-3, pp. 202–220, 2009.

[4] R. Roestenburg, R. Bakker, and R. Williams, *Akka In Action*. Manning Publications Co., 2016.

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

[6] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with SALSA," *ACM SIGPLAN Notices*, vol. 36, no. 12, pp. 20–34, 2001.

[7] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, "Modeling and verification of reactive systems using Rebeca," *Fundam. Inform.*, vol. 63, no. 4, pp. 385–410, 2004.

[8] H. C. Baker Jr and C. Hewitt, "The incremental garbage collection of processes," *ACM Sigplan Notices*, vol. 12, no. 8, pp. 55–59, 1977.

[9] NASA Engineering and Safety Center, "National highway traffic safety administration Toyota unintended acceleration investigation," NASA, Technical Assessment Report, January 18 2011.

[10] P. Koopman, "A case study of Toyota unintended acceleration and software safety," 2014. [Online]. Available: http://betterembsw.blogspot.com/2014/09/a-case-study-of-toyota-unintended.html

[11] E. A. Lee, *Plato and the Nerd — The Creative Partnership of Humans and Technology*. MIT Press, 2017.

[12] J. C. Corbett *et al.*, "Spanner: Google's globally-distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 8, 2013.

[13] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974, Conference Proceedings, pp. 471–475.

[14] G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," in *Information Processing*, B. Gilchrist, Ed. North-Holland Publishing Co., 1977, Conference Proceedings, pp. 993–998.

[15] J. B. Dennis, "First version data flow procedure language," MIT Laboratory for Computer Science, Report MAC TM61, 1974.

[16] W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the dataflow computational model," *Parallel Computing*, vol. 25, no. 13-14, pp. 1907–1929, December 1999.

[17] E. A. Lee and E. Matsikoudis, "The semantics of dataflow with firing," in *From Semantics to Computer Science: Essays in memory of Gilles Kahn*, G. Huet, G. Plotkin, J.-J. Lévy, and Y. Bertot, Eds. Cambridge University Press, 2009.

[18] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combinedlong-run average and worst-case performance analysis," in *Formal Methods and Models for Co-Design*, 2006, Conference Proceedings.

[19] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[20] S. A. Edwards and E. A. Lee, "The semantics and execution of a synchronous block-diagram language," *Science of Computer Programming*, vol. 48, no. 1, pp. 21–42, 2003.

[21] G. Berry and E. Sentovich, "Multiclock Esterel," in *Correct Hardware Design and Verification Methods (CHARME)*, vol. LNCS 2144. Springer-Verlag, 2001, Conference Proceedings.

[22] L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and P. Ölveczky, "PALS: Physically asynchronous logically synchronous systems," Univ. of Illinois at Urbana Champaign (UIUC), Report Technical Report, 2009.

[23] B. Zeigler, *Theory of Modeling and Simulation*. New York: Wiley Interscience, 1976, DEVS abbreviating Discrete Event System Specification.

[24] C. G. Cassandras, *Discrete Event Systems, Modeling and Performance Analysis*. Irwin, 1993.

[25] E. A. Lee and H. Zheng, "Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems," in *EMSOFT*. ACM, 2007, Conference Proceedings, pp. 114 – 123.

[26] X. Liu, E. Matsikoudis, and E. A. Lee, "Modeling timed concurrent systems," in *CONCUR 2006 - Concurrency Theory*, vol. LNCS 4137. Springer, 2006, Conference Proceedings, pp. 1–15.

[27] M. Lohstroh, M. Schoeberl, A. Goens, A. Wasicek, C. Gill, M. Sirjani, and E. A. Lee, "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*. ACM, 2019, pp. 152:1–152:4.

[28] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, "SCCharts: Sequentially constructive Statecharts for safety-critical applications," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, 2014, pp. 372–383.

[29] Y. Zhou and E. A. Lee, "Causality interfaces for actor networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–35, 2008.

[30] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Trans. on Software Engineering*, vol. 5, no. 5, pp. 440–452, 1979.

[31] D. Jefferson, "Virtual time," *ACM Trans. Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, 1985.

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

[33] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstr, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[34] E. A. Lee, J. Reineke, and M. Zimmer, "Abstract PRET machines," in *IEEE Real-Time Systems Symposium (RTSS)*, December 5 2017.

[35] F. Boussinot, "Reactive c: An extension to c to program reactive systems," *Software Practice and Experience*, vol. 21, no. 4, pp. 401–428, April 1991.

[36] F. Boussinot and R. de Simone, "The SL synchronous language," *IEEE Tr. on Software Engineering*, vol. 22, no. 4, pp. 256–266, April 1996.

[37] C. André, "SyncCharts: A visual representation of reactive behaviors," University of Sophia-Antipolis, Report RR 95–52, April 27 1996.

[38] L. Mandel, C. Pasteur, and M. Pouzet, "ReactiveML, ten years later," in *Int. Symp. on Principles and Practice of Declarative Programming (PPDP)*, July 14-16 2015, Conference Proceedings.

[39] R. von Hanxleden, "SyncCharts in C," Department of Computer Science, Christian-Albrechts-Universitaet Kiel, Technical Report Bericht Nr. 0910, May 2009.

[40] A. Schulz-Rosengarten, R. Von Hanxleden, F. Mallet, R. De Simone, and J. Deantoni, "Time in SCCharts," in *2018 Forum on Specification Design Languages (FDL)*, Sep. 2018, pp. 5–16.

[41] R. Von Hanxleden, T. Bourke, and A. Girault, "Real-time ticks for synchronous programming," in *2017 Forum on Specification and Design Languages (FDL)*. IEEE, 2017, pp. 1–8.

[42] F. S. de Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang, "A survey of active object languages," *ACM Computing Surveys*, vol. 50, no. 5, pp. 76:1–76:39, 2017.

[43] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[44] A. Prokopec and M. Odersky, "Isolates, channels, and event streams for composable distributed programming," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. New York, NY, USA: ACM, 2015, pp. 171–182.

[45] A. Prokopec, "Pluggable scheduling for the reactor programming model," in *Programming with Actors: State-of-the-Art and Research Perspectives*, A. Ricci and P. Haller, Eds. Springer International Publishing, 2018, pp. 125–154.