Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems

Marten Lohstroh

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-235
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html

December 21, 2020

Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems

by

Hendrik Marten Frank Lohstroh

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Edward A. Lee, Chair
Professor Gul A. Agha
Professor Alberto L. Sangiovanni-Vincentelli
Professor Sanjit A. Seshia

Fall 2020

Reactors: A Deterministic Model of Concurrent Computation
for Reactive Systems

Abstract

Reactors: A Deterministic Model of Concurrent Computation
for Reactive Systems

by

Hendrik Marten Frank Lohstroh

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Edward A. Lee, Chair

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. The same problem exists in other dominant concurrency models, such as threads, shared-memory models, publish-subscribe systems, and service-oriented architectures.

We propose "reactors," a new model of concurrent computation that combines synchronous-reactive principles with a sophisticated model of time to enable determinism while preserving much of the style and performance of actors. Reactors promote modularity and allow for distributed execution. The relationship that reactors establish between events across timelines allows for:

1. the construction of programs that react predictably to unpredictable external events;
2. the formulation of deadlines that grant control over timing; and
3. the preservation of a deterministic distributed execution semantics under quantifiable assumptions.

We bring the deterministic concurrency and time-based semantics of reactors to the world of mainstream programming languages through Lingua Franca (LF), a polyglot coordination language with support (so far) for C, C++, Python, and TypeScript. In LF, program logic is given in one or more of those target languages, enabling developers to use familiar languages and integrate extensive libraries and legacy code.

The main contributions of this work consist of a formalization of reactors, the implementation of an efficient runtime system for the execution of reactors, and the design and implementation of LF.

To Rusi and Luka.

Contents

List of Algorithms

List of Figures

List of Code Listings

List of Tables

Acknowledgments

This work was financially supported in part by the National Science Foundation (NSF) awards #1836601 (Reconciling Safety with the Internet) and #1446619 (Mathematical Theory of CPS), and the iCyPhy (Industrial Cyber-Physical Systems) research center supported by Avast, Camozzi Industries, DENSO International America, Inc., Ford, Siemens, and Toyota. It was also partly funded by the TerraSwarm Research Center, one of six centers administered by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

On the personal front, I owe much gratitude to my advisor, Edward A. Lee, who has been a tremendous source of inspiration and has given me freedom, opportunity, and invaluable mentorship. Collaborating with him on the design and implementation of Lingua Franca (LF) has been a true joy. I am humbled by his generosity, and in awe of his work ethic and the depth of his curiosity and skill. I could not have wished for a better advisor and I am honored to graduate as his student.

I would also like to thank Gul Agha, Alberto Sangiovanni-Vincentelli, and Sanjit Seshia. It is a great honor to have them on my dissertation committee. The work in this thesis is based to a significant degree on Gul Agha's work on actors. Alberto and Sanjit also influenced me profoundly with their ideas, and I have always felt encouraged by them throughout my years at Berkeley. Alberto's critical questions have been particularly helpful, and his suggestion to venture away from the well-trodden path of the actor abstract semantics has proved to be crucial in the development of reactors.

I thank Carl Hewitt, who originated the actor model of concurrent computation, for showing interest in my work and being supportive despite our differing perspectives on the role of determinism in concurrent systems.

The sustained work of Soroush Bateni, Hannes Klein, Shaokai Lin, Christian Menard, Alexander Schulz-Rosengarten, and Matthew Weber has been instrumental in the development of the LF language, runtime, and compiler toolchain. I thank Jeronimo Castrillon, Cong Liu, and Reinhard von Hanxleden for allowing and encouraging their students to devote their time to this project.

I am thankful to all my co-authors, and would like to give credit Andrés Goens for his very early involvement in the conception of reactors, and Íñigo Íncer Romeo for the substantial role he played in the formalization of reactors. The many conversations I have had with Yvan Vivid (formerly known as Chris Shaver) about models of computation have also helped set the stage for reactors. Christopher Gill, Martin Schoeberl, and Marjan Sirjani were part of key conversations that helped shape reactors. Related ideas expressed in the work on hybrid co-simulation that I collaborated on with David Broman, Fabio Cremona, Stavros Tripakis, and others, have also influenced this work. Ben Zhang helped me with the precursor of reactor-ts (when was still written in Flow). I thank Patricia Derler for helping me understand Ptides. Edward Wang has been assisting a work in progress to get reactors running on FlexPRET. Others with whom I have had insightful conversations about reactors

are: Ravi Akella, Baihong Jin, Matthew Milano, Mitar Milutinovic, and Mehrdad Niknami.

Shirley Salanio and Jean Nguyen were always helpful and graciously guided me through the bureaucracy of the EECS department. Christopher Brooks saved my bacon on multiple occasions and always looked out for me—he even pleaded me out of a citation for a traffic violation one day. I could always turn to Mary Stewart for assistance in logistical and hardware-related matters. Among the Berkeley faculty that I owe thanks to for their roles of support are: Edmund Campion, Paul Hilfinger, Chris Hoofnagle, Koushik Sen, and David Wagner.

The students, visitors, and faculty in the DOP Center, and the many wonderful people within EECS, but also in other departments, that I had the pleasure of interacting with, made my time as a grad student all the more interesting. The many conversations with Babak Ayazifar, Kris Pister, and other frequenters of the DOP kitchen, often brightened up my day. Even the late-night meetings I attended as a delegate of the Graduate Assembly I will miss. I will always be grateful for the friendships that I owe to Berkeley. I look back on many good times shared with Ilge Akkaya, Sebastian Conrady, Tommaso Dreossi, Roel Dobbe, Shromona Ghosh, Hokeun Kim, Stephen Moros, Adam Orford, Dax Ovid, and Alberto Tempia Bonda; as well as Machiel Blok, Florian Feuser, Kosuke Hata, Antonio Iannopollo, Gil Lederman, Hung Ngo, Stefan Pabst, Aviad Rubinstein, and their families.

As I wrote this thesis during the COVID-19 pandemic (and campus buildings were closed), I ended up doing most of the work in unusual places. One of them was Tilden Regional Park; another was "the shed" in the Bancroft Community Garden. Eventually I settled in The Office, a friendly co-working space in downtown Berkeley that remained accessible and gave me the peace and quiet I so desperately needed to complete this work.

I thank my parents, Jan and Yolanda Lohstroh, for providing help during challenging times and being supportive of my endeavors, no matter how far away they took me. My uncle, Frank Kooistra, deserves credit for having sparked my interest in computer science at an early age, which he did by putting books and hardware in my hands and telling me to RTFM. I also want to thank Tjitske Lohstroh, Eric Savelberg, Shamangi Kooistra, and Marten Kooistra, for their support and encouragement. I thank my dear friend Oscar de Boer, who stayed close in spite of being half a world away.

Lastly, and above all, I thank my wife Rusi Mchedlishvili, and my son Luka Marten Lohstroh. I could not have done this work without their love, their patience, and their unwavering support. I dedicate this work to them.

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

Edsger W. Dijkstra

Chapter 1

Introduction

This chapter draws from and expands on previously published work titled "Deterministic Actors" [142] that was co-authored with Edward A. Lee.

1.1 Motivation

While Alan Turing's "computing machines" [210] remain the bedrock of modern-day computing, many of the tasks performed by computers today are not to **compute** in the Church-Turing sense i.e., to produce a final result given some input. Rather, their purpose is to maintain an ongoing interaction with their environment. This is the case for embedded software that runs in electronics like modems and television sets, but also for operating systems that run servers, personal computers, and mobile devices, as well as for control software used in avionics, aerospace, and automotive applications. This broad class of so-called **reactive systems** [154] encompasses a substantial and growing portion of the computing systems we surround ourselves with.

There are two important aspects of reactive systems that not a part of Turing's model:

1. Concurrency; and
2. Time.

The work in this thesis incorporates these aspects as first-class concepts in "reactors," a deterministic model of concurrent computation for reactive systems. Before examining this new model, we explore how nondeterminism arises in a strongly related model of concurrent computation called actors. We also survey existing approaches to curbing nondeterminism in actor systems, some of which has served as foundations for reactors.

Concurrency

Concurrency has been a central theme in computing ever since the development of operating systems and computer networks that started in the early 1950s. Major work was done

in the late 70s by Hoare with his "communicating sequential processes" (CSP) [96] and in the early 80s by Milner with his "calculus of communicating systems" (CCS) [164] and Bergstra and Klop with their "algebra of communicating processes" (ACP) [22]. A later incarnation of CCS (also developed by Milner) called the π-calculus [165] adds expressivity by allowing configuration changes in the network between concurrent computations. Several variants of the π-calculus have been developed over the following decades, such as the ambient calculus [40] and join-calculus [72]. The focus of these process calculi and algebras is to enable formal reasoning about equivalences between processes (e.g., using bisimulation). What all of these models have in common is that they interpret concurrency as a matter of *interleaving*. That is, a situation where processes take turns.

The idea of "taking turns," however, presupposes some kind of arbitrator or centralized controller, and in systems that are composed of a mixture of computational and physical processes, an interleaving semantics fails to accurately describe the dynamics. After all, physical processes cannot be paused and may not be atomic. It is for this reason that in the field that studies **cyber-physical systems (CPS)** [125] an alternative approach to modeling concurrency, in which behavior is not necessarily reducible to an interleaving of processes, is often favored. This kind of model is also referred to as **real concurrency** [154] or *true concurrency* [184]. The actor model, introduced by Hewitt, Bishop, and Steiger [95] in the early 70s features real concurrency. Actors were given an operational semantics by Greif [84], a denotational semantics by Clinger [48], and a transition semantics by Agha [1].

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 systems, all of which we discuss in more detail in Section 1.2. A narrower definition, originally developed by Hewitt and his doctoral students [94, 2], appears in several popular software frameworks such as Scala actors [87], Akka [185], CAF [45] and Ray [168], and programming languages, like Erlang [9] and P [59]. Unlike various related dataflow models, the Hewitt actor model, as it is known, is **nondeterministic**, meaning that given an initial state and a set of inputs, a program can exhibit more than one behavior.

Actors are not alone in this. Most common software engineering approaches for expressing concurrent programs, including actors, but also threads [128], reactive programming [14], publish-subscribe systems [160], and even single-threaded event loops [6], make it very difficult construct deterministic programs. This is in stark contract with Turing model's of sequential computation, in which all programs are deterministic. But without a deterministic execution semantics, concurrent software tends to become intractable to rigorously test, let alone formally verify. We argue, therefore, that loss of determinism is a significant price to pay.

Actors have much in common with objects—a paradigm focused on reducing code replication and increasing modularity via data encapsulation—but unlike objects, actors provide a better model for concurrency than threads [128], the default model for objects. Indeed, each actor is presumed to operate concurrently alongside other actors with which it may

exchange messages. Objects, in contrast, are often designed assuming a single thread of control, and retrofitting them to be "thread safe" is challenging and error prone. The inherent concurrency of actors makes them ideal for programming reactive systems. However, the lack of any guarantees with respect to the ordering of messages and the absence of a notion of time make this model less useful for specifying systems in which repeatable behavior and/or timely execution are important.

Extra machinery can be introduced for the formal specification and analysis of systems composed of Hewitt actors. For instance, Real-time Maude [173], a timed rewriting logic framework and temporal model checking tool, has been applied to actors [61]. Similarly, the modeling language Rebeca performs analysis that uses a model checker to ensure that nondeterminism allowed in the model does not lead to behaviors that violate timing requirements [106]. Alternatively, constraints can be placed on actors' allowable behaviors so that they adhere to a stricter rule set, satisfying desirable properties (e.g., deadlock freedom, schedulability, bounded memory usage, and deterministic execution) by construction.

Time

Ren and Agha [183] have proposed giving actors a temporal semantics. As in our work, they assume a sufficiently well synchronized common physical time base shared by all actors, and they express timing requirements as constraints on message handling. Their work differs from ours, however, in that they build off a standard actor language, thereby inheriting its nondeterministic ordering of message handling, and they rely on separately imposing timing constraints to control the order when needed. In contrast, we use logical timestamps to define the order of message handling and ensure determinism.

Dataflow models are also closely related to the actor model. The (untimed) dataflow model has also been extended with formal contracts [215] that allow guarantees, e.g., for scheduling. There are timed models of dataflow [199], and even some structured approaches to use timing semantics in dataflow to execute time-critical applications in cyber-physical systems [80]. Fredlund et al. proposed timed extension of McErlang as a model checker of timed Erlang programs [62]. In this extension, a new API is introduced to provide the definition and manipulation of timestamps.

Even though many software applications are not particularly time sensitive, a semantic notion of time and the use of measurements of the passing of physical time can be powerful tools for achieving consistency in distributed systems [117, 223, 124]. Google's Cloud Spanner [50], for example, uses timestamps derived from physical clocks to define the behavior of a distributed database system; Spanner provides an existence proof that this technique works at scale. Moreover, logical time, as used in synchronous languages [20], for example, can provide a foundation for a deterministic semantics in concurrent programs.

Listing 1.1: Actor network that is deterministic under reasonable assumptions

```
1  actor X {
2      count = 1;
3      handler dbl(){
4          count *= 2;
5      }
6      handler inc(arg){
7          count += arg;
8          print count;
9      }
10 }
11 actor Y {
12     handler main {
13         x = new X();
14         x.dbl();
15         x.inc(1);
16     }
17 }
```

1.2 Background

Let us examine the problem of nondeterminism in the actor model and what can be done about it. We begin by illustrating the concern with a simple example, given in Listing 1.1. It uses a pseudo-code syntax that is a mashup of several of the concrete languages mentioned above. This code defines an actor class `X` that has a single integer state variable `count` that is initialized to 1. It has two message handlers, named `dbl()` and `inc()`. When invoked, these handlers will double and increment `count`, respectively.

The actor named `Y` with handler `main` creates an instance of `X` and sends it two messages, `dbl` and `inc`. 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 Listing 1.1 is deterministic under mild assumptions about message delivery and processing. First, we need to assume that messages are delivered reliably in the same order that they are sent. Since `dbl` is sent before `inc`, actor `x` will execute handler `dbl()` before handler `inc()`. Second, we need to assume that handlers are mutually exclusive.[1] 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.

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

Listing 1.2: Modifications of the code in Listing 1.1 yielding a nondeterministic program

```
1 actor Y {
2     handler main {
3         x = new X();
4         z = new Relay();
5         z.rly(x);
6         x.inc(1);
7     }
8 }
9 actor Relay {
10     handler rly (x){
11         x.dbl();
12     }
13 }
```

Consider now the seemingly minor elaboration shown in Listing 1.2. 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 an actor-oriented program. It 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 `inc()` is invoked first. (The final value of `count` will be 3 or 4.)

A similar example written in the concrete syntax of Ray [168] is shown in Listing 1.3. Ray extends the metaphor of remote procedure calls by integrating futures [16] 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 Listing 1.3 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. This mechanism could be used, for example, to make the program in Listing 1.3 deterministic. The `test` function could be replaced with the code in Listing 1.4. This code forces the main actor to block until the result of the invocation of `dbl()` is received before sending the `inc` 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.

Listing 1.3: A nondeterministic actor network in the syntax of Ray

```
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)
```

Listing 1.4: Modifications of the program in Listing 1.3 to make it deterministic

```
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)
```

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 Listing 1.2 to consistently print 3. One way would be to modify class `X` so that it *always* invokes `dbl()` before `inc()`, 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 `inc` 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 nondeterminism, 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 [171] 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 [111]. 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.

There exists a large body of prior work that can be framed as extensions of the Hewitt actor model that 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. We will explore these next.

Achieving Determinism

A system is **deterministic**[2] 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 `count = 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 [134]. 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. Newtonian mechanics, to give another example, provides a deterministic model of mechanical systems. The initial state of a system is the positions of its parts, the "inputs" are forces, and the "behavior" is motion in a three-dimensional space over a time continuum. Quantum mechanics, on the other hand, cannot predict the exact location of a particle in space, only the probability of finding it at different locations [30]. Thus, if "behavior" is motion (change of position over time), then the quantum mechanical model of the system is not deterministic. The same physical system, therefore, is deterministic or not depending on the model.

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.

[2]Following Milner [166], some use the term *determinacy* to refer strictly to input/output relations that are functions in the mathematical sense of the word, and *determinism* to include some notion of behavior (such as a particular sequence to computational steps taken to yield a certain result). We use these terms interchangeably.

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 [50], 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 Listing 1.2 and 1.3. 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 `inc()`, *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.

Coordination for Determinism

Let us focus on the actor network sketched in Listing 1.2. Since actor `Y` first sends a message that has the eventual effect of doubling `count` of actor `X` and then sends a second message to increment `count` 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 concurrently executing actors 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) [103]. In 1977, Kahn and MacQueen showed that a very simple execution policy using blocking reads guarantees such determinacy [104]. Using the Kahn-MacQueen principle, actor `X` in Listing 1.2 could be replaced with `X_KPN` (for Kahn Process Network), shown in Listing 1.5. 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 Listing 1.5 perform such blocking reads. That code ensures that doubling `count` will occur before incrementing it even if actor `Y` sends its output messages in opposite order.

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,

[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.

Listing 1.5: Variant of `X` in Listing 1.2 to encode design intent using blocking reads

```
1 actor X_KPN  {
2   handler main {
3     count = 1;
4     await(dbl);
5     count *= 2;
6     arg = await(inc);
7     count += arg;
8     print count;
9   }
10 }
```

Listing 1.6: Modification of actor `Relay` in Listing 1.2 to filter messages

```
1 actor Relay {
2   handler rly (X x) {
3     if (some condition) { x.dbl(); }
4   }
5 }
```

as shown in Listing 1.6. 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 [57, 170]. Modern versions use carefully crafted notions of "firing rules" [129], 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 components 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 [205]. Although dataflow models are generally untimed, there have been some efforts to augment them with limited temporal semantics [80]. Kahn process networks and most dataflow models lack the notion of a message handler, something that appears in most modern realizations of Hewitt actors. Although message handlers are merely a convenience, for complex actors, they significantly simplify the design.

Another family of coordination techniques that can deliver deterministic execution uses the synchronous-reactive (SR) principle [20]. 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 [66] 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 [24]. 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) [195].

A fourth alternative, and the one that is the focus of this thesis, is based on discrete-event (DE) systems, which have historically been used for simulation [221, 41], 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 [132]. 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 ultrametric spaces rather than complete partial orders, but this semantics similarly guarantees determinism [140].

1.3 Contributions

The main contributions of this work consist of:

1. a formalization of reactors—a deterministic model of concurrent computation for reactive systems;
2. the implementation of an efficient runtime system for the execution of reactors; and
3. the design and implementation of Lingua Franca (LF)—a polyglot coordination language based on reactors.

These contributions are meant to enable a methodology for the design and implementation of concurrent systems that are deterministic by default. Our approach strives for understandable concurrency, improved analyzability, and scalable testing. Central to our programming model is a semantic notion of time that allows for the existence of multiple timelines and makes an explicit distinction between logical time and physical time. We leverage this distinction to allow for the formulation of deadlines, as well as the injection of sporadic events into a running system such that it is able to provide deterministic responses to external stimuli. The relationship between logical time and physical time that reactors establish can also be exploited to implement a fully distributed coordination scheme that preserves determinacy under quantifiable assumptions.

We show that a runtime environment implemented in a language that support threads can automatically exploit parallelism in reactor programs. We also demonstrate that it is possible execute reactors under an earliest-deadline-first scheduling policy. Our preliminary performance evaluation suggests that the determinism of reactors does not come at the cost of a performance loss when compared to ordinary actors.

The explicit data dependencies of reactors allow for a "black box" coordination approach which is the key enabler of the polyglot nature of LF Using LF, it is possible to integrate reactors with legacy software and the extensive libraries that are an important factor in the popularity of existing programming languages. LF is also capable of generating "federated" programs which consist of reactors mapped to across hosts that communicate over a network.

1.4 Related Work

Actors

The work in this thesis is closely related to languages and frameworks that evolve around the actor model [94, 2]. Actor based languages include Erlang [9], Scala actors [87], Salsa [212], Rebeca [198], and P [58]. Noteworthy actor frameworks are Akka [185], Ray [168], and CAF [45]. The flexibility of actors allows building systems that are scalable and resilient to failures, but this comes at the cost of inherent nondeterminism, which poses challenges to verify the correctness of actor systems. Rebeca provides a formalism and model checking techniques for analyzing and verifying actor networks. While this can improve confidence in a correct implementation, the programmer is still responsible for finding this correct implementation. P goes a step further in that it also has an efficient runtime system and compiler that generates code with reasonable performance. P now also has a verifier based on the UCLID5 modeling and verification language [193].

The concept of "reactive isolates" [178] (coincidentally, later also called "reactors") was introduced to modularity combine different communication protocols inside the same actor (realized in the Scala-based Reactors.IO framework [176]). 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.

Active Objects

Also related are a family of so-called "active object" languages [29], which 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 [128] is sparse in some of these languages. Very recent work by Henrio, Johnson, and Pun [91], studies the problem of active objects with guaranteed deterministic behavior, which they relate to the satisfaction of confluence properties between execution steps. They propose a core language for active objects in which well-

typed programs exhibit deterministic behavior. However, there appears to be no concrete programming language implementation that leverages these ideas yet.

Content-addressable Memory

The "generative communication" paradigm of Linda [81] aims to unify the notion of process creation and data exchange by encapsulating them in the same operation—the creation of a tuple—and carrying out the distribution of tasks and delivery of data in an abstract middleware layer. While this approach achieves a nice separation of concerns between computation and coordination, it places the order in which events are observed beyond the programmer's control. As such, a shared memory model provides very little support to the programmer for achieving determinism. Tuple space may be thought as a form of distributed shared memory [167].

Synchronous Languages

The use of synchronous-reactive principles to deterministically coordinate concurrent software has a long history, with notable contributions like Reactive C [33], SL [34], SyncCharts [7], and ReactiveML [153]. A modern variant of SyncCharts, SCCharts [88], composes finite state machines under a synchronous semantics. It has been recently augmented with a semantic notion of time [188] based on the concept of dynamic ticks [213, 189]. Like reactors, components can inform the scheduler at what logical time to trigger reactions.

Synchronous languages, such as Esterel [26], Lustre [86], and SIGNAL [21], make an abstract notion of time an essential part of the language. SIGNAL and Multiclock Esterel [24], explicitly support a multiplicity of abstract timelines. SIGNAL supports asynchronous actions and nondeterministic merging of signals. Some care is required when comparing our work to these efforts, however. We use the term "clock" in a more classical way as something that measures the passage of physical time. In the synchronous language use of the term "clock," a sequence of events sent from one reactor to another has an associated "clock," which is the sequence of tags associated with those events. Since these clocks can all be different, LF supports at least the multiplicity of timelines like those in Multiclock Esterel. A federated execution of LF also has the capability of decoupling logical time advance, so despite our tags coming from a totally ordered set, LF achieves properties similar to the polychrony of SIGNAL. LF can even accomplish the nondeterminism of SIGNAL by using physical connections. Like LF, SIGNAL can be used effectively to design distributed systems [77]. A major difference, however, is that LF is a coordination language, with the program logic expressed in a target language (C, C++, Python, or TypeScript), whereas SIGNAL is a complete standalone programming language.

Dataflow and Process Networks

The embedded systems community commonly uses variants of the actor model with deterministic semantics such as dataflow models [57, 27, 120] and process networks [103, 121]. The fixed graph topologies inherent to these models, enable improved static analysis and optimization [79], but the static topology also limits flexibility and the application's capability to react to external events.

Other works that follow a deterministic-by-construction approach but are more focussed on parallel computing are LVars [115], FlowPools [177] and isolation types [37]. LVars ensure determinism by allowing only monotonic writes and "threshold" reads that block until a lower bound is reached. FlowPools are a data structure for composable deterministic parallel dataflow computation through the use of functional programming abstractions. Isolation types let programmers declare what data they wish to share between concurrently executing tasks, and those tasks fork and join isolated revisions of the shared data.

Reactive Programming

Reactors have an overlap with the reactive programming paradigm. In reactive programming language runtimes, programs are also internally represented by a dependency graph for the purpose of automatically (re)computing parts of a program whenever values change. The reactive programming community is mostly focussed on asynchronous, event-driven, and interactive applications. Sometimes a framework or language is already called "reactive" if it implements the observer pattern [78]. A wide range of reactive software technologies is available [14] including programming frameworks like ReactiveX [161] and Reactors.IO [176] as well as language-level constructs like event loops [207], futures [16], and promises [73]. All synchronization in actor and reactive programming frameworks needs to be added explicitly by the programmer. Futures are commonly used to avoid the explicit use of continuation messages, which gives the "feel" of imperative code but does not prevent programming errors due to nondeterminism. Finding such errors in reactive systems is particularly difficult [17, 148]. Even more problems arise if languages, frameworks and libraries do not enforce the underlying model and invite the programmer to break its semantics [203].

Hardware Description Languages

Another family of languages that are related to LF are hardware description languages (HDLs) such as Verilog [206] and VHDL [8], which can be used to model a digital system at many levels of abstraction, ranging from the algorithmic level to the gate level. These concurrent programming languages follow both the dataflow and reactive programming paradigm. They are mostly aimed at electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits, but they can be used as a general-purpose parallel programming language as well. Verilog is nondeterministic [64]

in ways that VHDL is not[4] [150] as it updates its internal state and advances to the next tick only after having handled all pending requests. Unlike reactors, the outputs of components in VHDL often have to be manually brought into alignment with "delta cycles" (akin to microsteps in our superdense time model). This is because outputs are always produced with a (microstep) delay in VHDL, whereas reactors can produce output logically instantaneously. Also noteworthy are the guarded atomic actions of another HDL called Bluespec [11], which bear resemblance to the reactions of reactors.

Frameworks for System-level Modeling

System-level modeling frameworks such as SystemC [135, 202], and the related SpecC [76] are closely related to hardware description languages. Programs written in these using these frameworks compile down to executable programs that implement a discrete-event model much like reactors do. Where LF is intended for the coordination of software components in deployed systems, the goal of these frameworks is to aid tasks like design space exploration and performance modeling.

Modeling and Simulation Tools

A lot of valuable lessons learned in the Ptolemy project [179] and from related modeling and simulation tools such as Simulink [52] and LabVIEW [28] have been reapplied in the design of and implementation of reactors and LF. The influence that the Ptolemy project has had on this work is difficult to overstate. There is a significant overlap between the reactor model and the discrete-event domain in Ptolemy II, but there are important differences. The most fundamental difference is that reactors are a departure from the actor abstract semantics [209] that form the blueprint of all models of computation implemented in Ptolemy II. Where a Ptolemy actor has a single "fire" function, reactors can have multiple reactions, some, all, or none of which could execute at any given logical time (depending on which triggers are present). Reactors and Ptolemy actors share the notion of ports, but the notion of actions (and their physical or logical "origin" that lets their events be linked to either a physical or logical timeline) are unique to reactors. The focus of reactors is on performance and interaction with the physical world, whereas Ptolemy is aimed at the modeling and simulation of cyber-physical systems.

Accessors

Reactors could be viewed as a continuation of the work on accessors [119, 36], which are actors that interface asynchronous atomic callbacks through a deterministic coordination layer based on discrete events [145]. Reactors can fulfill the same role as accessors and

[4] Strict determinism was lost with the '93 revision of the VHDL standard, which introduced shared global variables [98].

serve as proxies for remote services, sensors, or actuators. The reactor model's notion of actions and reactions resolves an awkwardness that exists in accessors due the fact that their functionality is broken down in a set of distinct handlers even though they still have a single "fire" function that causes all handlers to be invoked when the component fires, making it difficult to schedule the execution of a specific handler. Accessors also feature runtime mutations, but the dynamic substitution mechanism only takes into account whether the port types match, not whether the modification could introduce causality loops, for example. In [102], accessors were augmented with labeled logical clock domains (LLCDs), which allow asynchronous events to be scheduled relative to the last-known time in a particular clock domain. In reactors, this would be synonymous with asynchronously scheduling an event on a logical action, which we explicitly prohibit as this could cause logical time to lag arbitrarily far behind of physical time, or allow attempts to schedule events in the past with respect to the current logical time. In a federated reactor program such behavior could also lead to one federate blocking the advancement of logical time in other federates.

Real-Time Languages

Time naturally plays an essential role in application design for real-time systems [201, 112]. Many languages such as Real-Time Euclid [110], Ada [38], or Real-time Java [208, 214] provide support for modeling temporal as well as behavioral application aspects. Commonly, such languages focus on time-predictability and are often limited to specifying schedules of periodic tasks, whereas the primary focus of LF is coordination based deterministic coordination of periodic as well as sporadic tasks. Ada is one of the few languages other than C that have seen a considerable degree of adoption in avionics, air traffic control, railways, banking, military and space technology [70]. Another notable exception is the synchronous language SCADE [23]. Ada's tasking system is based on an event-driven scheduling model, but unlike that of reactors, it is nondeterministic [15]. While LF has potential as a language for designing hard real-time properties, we consider this a secondary goal.

Like LF, Timed C [172] has a logical time that does not elapse during the execution of a function (except at explicit "timing points"). Moreover, like LF, priorities are inferred from timing information in the program. The deadlines of LF are all "soft deadlines" in the terminology of Timed C, meaning that the tasks are run to completion even if they will lead to a deadline violation. It would be useful further work to realize the "firm deadlines" of Timed C, but these require the use of low-level C primitives `setjmp` and `longjmp`, and it is not clear that it is possible to provide these in our polyglot approach.

Giotto [92] provides an abstract programming model for the implementation of embedded control systems with hard real-time constraints. In Giotto, platform-independent concerns (functionality and timing) are separated from platform-dependent ones (mapping and scheduling). Unlike LF, Giotto is a purely task-based model. It is augmented with modes, which allows for more flexibility, but it is not equipped with the ability to handle sporadic events, which reactors are able to do via physical actions.

The logical execution time (LET) paradigm [108], where logical time delays are used to "mask" physical delays in the system so that all delays are predictable and exact, is fully compatible with reactors.

Verification of Timing Properties

One of the distinguishing features of reactors is its model of time. There exist formalisms that also embrace a multiplicity of time lines in parallel and distributed systems. The MARTE profile of UML, and its Time Model and CCSL (Clock Constraint Specification Language) [152] specify constraints among instants in a multiplicity of clocks. TimeSquare analyzes systems of constraints in CCSL [56]. CCSL can be used for embedded systems with distinct clocking mechanisms [174]. (such as time-driven combined with crankshaft-rotation driven). TESL (Tagged Events Specification Language), which is based in part on CCSL, like LF, uses explicit tags and ensures determinism [31]. Neither TESL nor CCSL is a programming language, but rather a language for modeling timing relationships. They could prove useful for analyzing LF programs.

1.5 Outline

The remainder of this thesis is organized as follows:

- Chapter 2 provides a brief introduction of reactors followed by a formalization of reactors. This chapter provides all the algorithms that are involved in the execution of reactors.
- Chapter 3 introduces Lingua Franca by explaining its basic syntax and discussing examples. At the end of this chapter we show that any reactor can be modeled as a strictly contracting function. This allows us to conclude that the behavior of any reactor at any logical time can be expressed in terms of a unique fixed point, and therefore is deterministic.
- Chapter 4 discusses the use of physical actions, the distinction between physical and logical actions, and the workings of deadlines and logical time delays. It explains in more detail how reactors expose parallelism, how earliest-deadline-first scheduling can be achieved, and what can be done to improve performance. We also show how reactors can be extended with support for subroutines. This chapter concludes with a preliminary evaluation that compares our performance against the popular actor framework Akka over a small subset of the Savina [99] benchmark suite.
- Chapter 5 shows how reactors can execute in a federation that spans multiple hosts. We discuss a centralized coordination method fashioned after HLA [114] and a decentralized coordination method based on Ptides [223].
- Chapter 6 provides conclusions and discusses avenues for further work.

All problems in computer science can be solved by another level of indirection.

David Wheeler

Chapter 2

Reactors

This chapter draws from and expands on previously published work titled "Reactors: A deterministic model for composable reactive systems" [144] that was co-authored with Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli.

Reactors, first described in [146], can be thought of as deterministic actors composed of reactions. Reactions bear resemblance to the message handlers of actors, except rather than responding to messages, reactions are triggered by discrete events and may also produce them. An event relates a value to a tag. A tag represents the logical time at which the event was released into the system. Reactions have access to state shared with other reactions, but only within the same reactor. Events are the only means by which reactors can communicate with one another.

Where message handlers in actors are invoked in no particular order, the order in which reactions in reactors can occur is subject to specific constraints. Events are always observed in tag order. Events with identical tags are logically simultaneous. An event can trigger a reaction, but a triggered reaction does not execute before all events with the same tag that it can observe have been produced. This means that any reactions responsible for producing such events are forced to execute prior (should they be triggered at that logical time). Reactors can be classified as a "sparse synchronous model" [63] as the synchronous-reactive interactions that occur at any particular logical time may be limited to isolated parts of the system. When a reaction executes, it has exclusive access to the reactor's state, and for any two reactions of the same reactor that are triggered by events with the same tag (or one and the same event), the order in which they execute is predefined. Because of these constraints, reactors react deterministically to inputs, making it possible to verify the correctness of their behavior through testing.

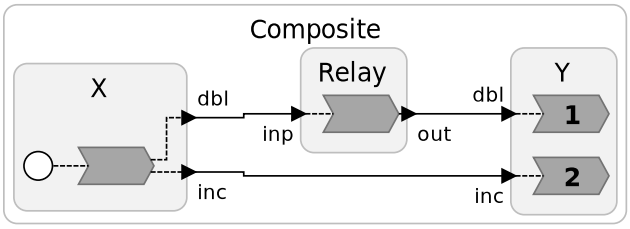


Figure 2.1: A reactor implementation of the introductory example. Reactor X has a startup reaction that produces an event on output ports dbl and inc. The second reaction of Y, triggered by input port inc, cannot execute 1) before Relay has reacted; and 2) until after the first reaction of Y has executed in case it was triggered by an event on dbl.

2.1 Ports, Hierarchy, and Actions

The term "reactors" is a nod to Hewitt and Agha's actors [95, 1] (actors, revisited), but also to the synchronous reactive programming paradigm that underpins languages like Esterel [26], SIGNAL [21], Lustre [86], and their derivatives. Different from Hewitt actors, reactors do not directly refer to their peers. Reactors have named (and typed) ports that allow them to be connected to other reactors (depicted by black triangles in Figure 2.1). An event produced by one reactor is only observed by other reactors that are connected to the port on which the event is produced. Events arrive at input ports, and reactions produce events via output ports.

The extra level of indirection implied by ports enables a hierarchical design where a reactor that contains other reactors, such as the one named Composite in Figure 2.1, has access to dependency information that makes it possible to enforce ordering constraints that preserve determinacy. The Composite reactor in Figure 2.1 contains one instance of each of the three other reactors and defines how their ports are connected. In this example, there is only one composite, but composite reactors themselves can also have input and output ports and can be contained by (and connected to) other composites. There is one exception: top-level reactors are not allowed to have inputs or outputs. The containment hierarchy of reactors also serves as a scoping mechanism for ports, imposing constraints on the kinds of connections that can be drawn. Specifically, connections are not allowed to traverse more than one level of hierarchy. For instance, it is possible to connect the input port of a container to the input port of a contained reactor, but not to any input ports embedded deeper in containment hierarchy.

Reactors also feature a special variant of ports called actions. Unlike ports, actions are not visible to other reactors and it is not possible to connect to them. Actions are used for scheduling events that trigger reactions of the same reactor at a future logical time. Actions are also used as a synchronization mechanism between the logic inside reactions, which execute at well-defined logical time instants, and asynchronous events originating from the environment, such as data being reported from a sensor or a message being received through a network interface. Such sporadic external events get assigned tags in a way that ensures

determinacy, in the sense that once a tag has been assigned, the response of the reactor program is well defined.

In addition to user-defined actions, each reactor has two distinguished triggers: one called "startup," (represented by the circle in Figure 2.1) which is present only at the very first time instant of a reactor's execution, and another called "shutdown," which signals the end of a reactor's execution.

2.2 State Variables

Reactions may share state variables with other reactions in the same reactor. It is this capability that is the prime motivator for bundling multiple reactions in a single reactor. A shared resource may simply be a variable, but it could also be a socket, or a physical device. 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 [89], which are an extension of Harel's Statecharts [90] that allows arbitrary sequential reads or writes of shared variables during a synchronous-reactive tick. Because reactors do not share state among one another, if two distinct reactors receive logically simultaneous messages, then their reactions may be invoked in parallel unless there exists a connection between the two reactors that requires the upstream reactor's reaction to execute first.

2.3 Connections

The usage of ports and connections to establish explicit communication channels between actors—such as is done in Ptolemy II [179]—readily exposes dependencies that are difficult to infer in a setting where actors address each other directly. But even such an explicit communication topology does not reveal all dependency information required to make well-informed scheduling decisions that honor data dependencies.

Internally, actors may also establish dependencies between ports through their application logic (i.e., inside their handlers). The most conservative approximation of application logic induced dependencies would assume that all outputs of an actor depend on all inputs of that actor, but that may lead to the false flagging of potential problems such as zero-delay feedback loops or deadlock situations. In Ptolemy II, actors are equipped with causality interfaces [224] to report dependencies more accurately. Those dependencies, however, need to be declared by the programmer or inferred on the basis of code analysis. Reactors, on the other hand, make causality interfaces an integral part of component definitions, by breaking down their functionality into reactions, each of which is subject to simple lexical scoping rules that limit access to input and output ports, thereby eliminating dependencies between ports that are out of scope.

A major advantage of this approach is that the causality interface of a reaction is always *complete*. If a dependency exists, then it must be reflected in the interface definition. Using this scheme, a programmer cannot forget to declare a dependency without breaking the program, and effort associated with declaring dependencies incentivizes the programmer to only declare dependencies necessitated by the logic in the reaction. This promotes the goal of imposing the fewest constraints necessary to preserve causal consistency during the execution of a reactor program while leaving maximum freedom to the runtime scheduler to exploit parallelism in the program—all while treating the functionality of a reaction as a black box. This allows a schedule to be devised purely based on dependency information, although there is a cost. If a reaction declares that it reads an input, for example, then it may only be executed after that input is known. If it then does not actually read the input, due to a data dependency in the reaction body, then the constraint was unnecessary.

Statically declaring the dependencies comes at the cost of a slight loss in the accuracy of the reporting of causal dependencies, but it facilitates the polyglot nature of LF the reactor-oriented coordination language we discuss in Chapter 3. In LF, the implementations of reactions are given in verbatim target code that is not even parsed, much less analyzed. While it would be possible to infer the declared input/output dependencies through static analysis, whenever 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. Through LF, a variety of target languages can be supported by the reactor 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.

2.4 Example: Drive-by-wire System

To illustrate how reactors behave, let us return to the "unintended acceleration" problem mentioned in the introduction and consider a power train of an electric vehicle implemented using reactors. Our example, illustrated using the diagram in Figure 2.2, implements a so-called drive-by-wire system. In most modern road vehicles there is still a mechanical coupling between brake pedal and brakes, but so-called "brake-by-wire" designs have started to appear in cars in the recent years. These modern designs can improve the braking efficiency and stability of the vehicle [219]. The six reactors contained in the PowerTrain reactor 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 [35], each reactor in the figure (represented by a box with rounded corners) 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; other dependencies (through ports and actions) are

represented by dashed lines.

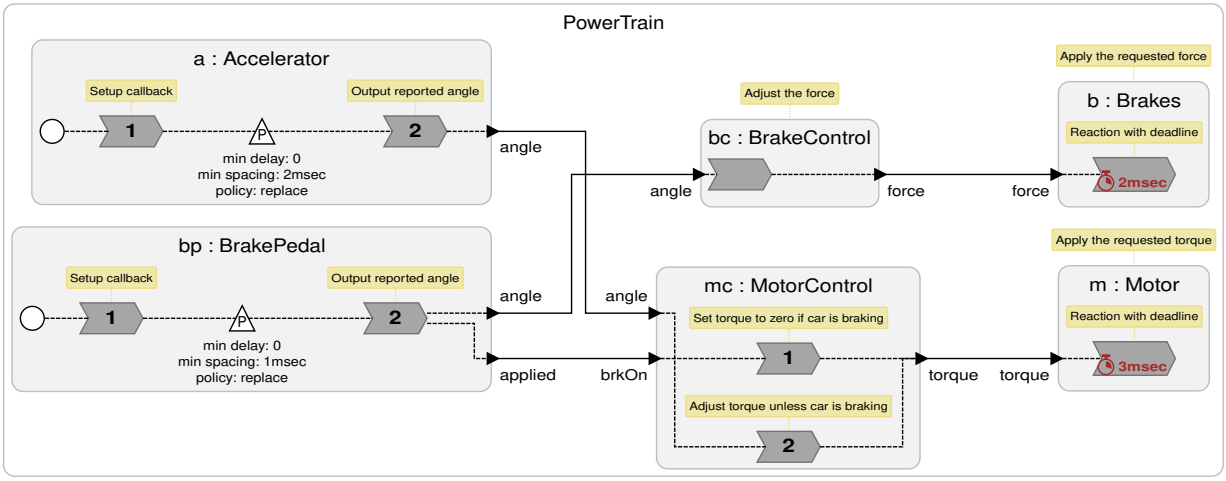


Figure 2.2: A reactor that implements a simplified power train control module.

Consider the bp (brake pedal) reactor, in Figure 2.2, which is used to signal the braking demand. We assume that updates from the pedal are reported via an interrupt, which enables an interrupt service routine (ISR) that schedules a physical action (represented by a small triangle labeled with a P) without further delay. This internal action triggers a reaction that sets the value of the angle and applied output ports. In order to avoid overwhelming the system, the physical action specifies a minimum spacing of 1 ms, which means that subsequent invocations of this reaction are always at least one millisecond apart. The values angle and applied, if present, are propagated to bc (brake control) and mc (motor control), respectively. Notice that bp only has to set applied 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. This sparsity of events is characteristic of reactor systems and other "sparse synchronous" models [63]. Eliminating "redundant" events has the advantage of reducing system load and making execution traces easier to comprehend.

Let us now consider the mc reactor, which has two reactions. We interpret the number associated with each reaction as its priority; this way, we obtain an execution order in case both brkOn and angle are present at the same logical time. The first reaction, mc.1, is triggered by brkOn; it updates the state of the reactor to reflect whether the brake pedal is currently pressed or released. If the brakes are being applied, then it adjusts the torque to zero. The second reaction, mc.2, is triggered by the angle input; it checks a state variable to see whether the brakes are applied, and *only* if this is not the case, sets the torque output in correspondence with the requested angle.

The design of a (accelerator) is identical to that of bp except for the larger minimum spacing of 2 ms, limiting the frequency at which a can produce events to 500 events per second.

The design assures that when the accelerator pedal is stuck or reports faulty readings, the car will still slow down in response to the break pedal being pressed; the motor 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.

Finally, reactors can be subjected to deadlines, which are elaborated on in more detail in Section 4.1. Two deadlines are present in Figure 2.2: a 2 ms deadline marked at the reaction of b (brakes) and a 3 ms deadline marked at the reaction of m (motor). Conceptually, these deadlines specify end-to-end physical time delays in the system between the occurrence of events and reactions triggered by those events. An event may originate from a physical action that is scheduled in response to the arrival of a sensor reading (e.g., bp in our example), and the triggered reaction may be driving an actuator (e.g., reactor b). The 2 ms deadline in our example simply states that reaction in b should start no later than 2 ms past the physical time at which bp reported a new angle.

2.5 Formalization

In this section we formalize the concept of reactors and specify their behavior as it is guided algorithmically by the runtime environment that performs their execution. The provided algorithms ought to be interpreted as an abstract reference implementation, and, as such, delineate the dynamic semantics of reactors. For readers who prefer to read programming language syntax and code examples over mathematical notation, it may be advisable to skip over this material and continue reading at the start of Chapter 3. Specific parts of the formalization that offer clarification of aspects not fully covered in the remaining chapters will be referred back to, making it easy to consult them whenever a more formal explanation is preferable. Appendix A also provides a summary of our model as a quick reference for looking up the meaning of symbols we use in our notation.

Some 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 $x = (a, b)$, 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 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: $\varepsilon \in V$ is called the **absent value**.

Notions of Time

Reactors use a **superdense model of time** [133, 151]. In this model, a time instant is represented by a tag [123]. A 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. Formally, the set of tags $\mathbb{G} = \mathbb{T} \times \mathbb{N}$, where $\mathbb{T}$ is order-isomorphic with the natural numbers and $\mathbb{N}$ is the set of non-negative integers. Two tags are equal if and only if both their time value and microstep index are equal. We define a total order on $\mathbb{G}$ lexicographically: if $(t, m), (t', m') \in \mathbb{G}$, we say that $(t, m) < (t', m')$ if and only if $(t < t') \vee (t = t' \wedge m < m')$. $\mathbb{G}$ has an addition operation that operates element-wise. Using an integer representation for time ensures that addition is associative [51], which is not normally the case when using floating-point representations[1]. Given a tag $g = (t, m)$, we can extract the time value and the microstep using the conventional projection operator π, where $\pi_1(g) = t$ and $\pi_2(g) = m$.

Definition 1 (Logical time)**.** ***Logical time*** *is a monotonically increasing sequence of* ***tags*** *of the form $g = (t, m)$, where t is referred to as the* ***time value*** *and to m as the* ***microstep index****.*

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.*

While we use logical time to track the progress our computation, we use physical time to understand the order of events observed in the physical world and order these events with respect to events on our logical timeline. We assume a background Newtonian time $\tau \in \mathbb{R}$, but in our notation we will only refer to time values $T \in \mathbb{T}$ that represent imperfect measurements of it. We use the same set for time values drawn from physical clocks as the time values that originate from logical clocks because we are interested in cyber-physical systems, which conjoin the dynamics of the physical world with that of the software. This allows us to take the current physical time T and place it on our logical timeline by converting it to a tag $g = (T, 0)$. Likewise, we can take a tag $g = (t, m)$ and compare its time value $\pi_1(g)$ to T.

Definition 2 (Physical time)**.** ***Physical time*** *refers to a time value $T \in \mathbb{T}$ that is obtained from a clock on the execution platform.*

[1] Recent work by Ahrens, Demmel, and Nguyen [4] describes a method that achieves reproducible summation independent of summation order using only standard floating-point operations, but at a $7x$ performance cost. New floating-point operations described in the IEEE Floating Point Standard-2019 [97] can be used to reduce the cost of their algorithm.

Tagged Events

Definition 3 (Event)**.** *An **event** e is defined as a list $e = (a, v, g)$, where $a \in \Sigma$ is called the event's action, $v \in V$ its **value**, and $g \in \mathbb{G}$ the tag. Events inherit an order from their tags. If e and e' are events, we say that $e < e'$ if and only if $g(e) < g(e')$. Finally, e and e' are **logically simultaneous** if and only if $g(e) = g(e')$.*

Given an action a, we define $\mathcal{T}(a)$ to identify the reactions that are triggered by a (see Table 2.1).

Events	
Event instance	$e = (a, v, g)$
Event action	$a \in A$
Event value	$v \in V$
Event tag	$g \in \mathbb{G}$
Triggered reactions	$\mathcal{T}(a) = \{n \in \mathcal{N}(C(a)) \mid a \in \mathcal{T}(n)\}$

Table 2.1: A formal model of events.

Reactors

We now proceed to define reactors.

Definition 4 (Reactor)**.** *A **reactor** r is a list $r = (I, O, \mathcal{A}, S, \mathcal{N}, \mathcal{M}, \mathcal{R}, \mathcal{P}, \{\bullet, \diamond\})$, where*

1. *$I \subseteq \Sigma$ is a set of **inputs**,*
2. *$O \subseteq \Sigma \times V$ a set of **outputs**,*
3. *$\mathcal{A} \times V$ a set of **actions**,*
4. *$S \subseteq \Sigma \times V$ a set of **state variables**,*
5. *$\mathcal{N}$ a set of **reactions**,*
6. *$\mathcal{M} \subseteq \mathcal{N}$ a set of **mutations**,*
7. *$\mathcal{R}$ a set of **contained reactors**,*
8. *$\mathcal{P} : \mathcal{N} \to \mathbb{P}$ the **priority function**, and*
9. *$\{\bullet, \diamond\}$ distinguished triggers called **startup** and **shutdown**, respectively.*

Given two reactors r and r', the sets $I(r)$, $O(r)$, $\mathcal{A}(r)$, $S(r)$, $I(r')$, $O(r')$, $\mathcal{A}(r')$, 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 $\mathcal{M}(r')$.

While input ports, output ports, and state simply bind identifiers to values, actions are more elaborate because they need to provide the runtime scheduler with additional information.

The reactor tuple distinguishes between *reactions* and *mutations* which are a subset of reactions that are capable of changing the internal structure (i.e., the contents of $\mathcal{R}$ and $\mathcal{N}$) of the containing reactor. Because such structural changes have further-reaching consequences than the mere triggering of reactions, additional dependencies are implied by mutations (see Section 2.6).

Reactors can contain other reactors, which are listed in reactor set $\mathcal{R}$. We use the following definition to navigate the hierarchy of a reactor.

Definition 5 (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: 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 actions: 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* $C(a) = r''$ *if and only if* $a \in \mathcal{A}(r'')$*. Similarly, the function* C *is well-defined here since all the relevant sets are pairwise disjoint for two distinct reactors.*

Remark 2 (Hierarchy)**.** *We define an* ***atomic reactor*** *as above, with an empty contained reactor set* $\mathcal{R}$*. 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* $n - 1$*. Moreover, the reactor set of a degree-*n *reactor contains at least one reactor of degree* $n - 1$*.*

Using $\mathcal{P}$, the reactor imposes an order on its constituent reactions to serialize the execution of simultaneously triggered reactions. It does this by mapping all contained reactors $\mathcal{R}$ to $\mathbb{P}$, which is defined as follows.

Definition 6 (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****,* $\mathbb{P}$*, is given by* $\mathbb{P} = \mathbb{Z}^- \cup \mathbb{Z}^+ \cup \{*\}$*. The set* $\mathbb{P}$ *is 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 of the same reactor to be executed in parallel if is statically known that they do not touch the reactor's state.

Remark 3 (Connections)**.** *Notice that* ***connections*** *are not modeled explicitly in this formalization. Instead, we represent connections with reactions whose only purpose is to relay the value from one port to another. These so-called* ***relay reactions*** *are assigned the priority* $*$*, meaning they can execute concurrently with other reactions in the containing reactor. To preserve determinacy, it must be checked that if there exist two or more reactions that*

Reactors	
Reactor instance	$r = (I, O, \mathcal{A}, S, \mathcal{N}, \mathcal{M}, \mathcal{R}, \mathcal{P}, \{\bullet, \diamond\}) \in R$
Set of input ports for r	$I(r) \subseteq \{p \in P \mid C(p) = r\}$
Set of output ports for r	$O(r) \subseteq \{p \in P \mid C(p) = r\}$
Set of actions for r	$\mathcal{A}(r) \subseteq \{a \in A \mid C(a) = r\}$
Set of state variables for r	$S(r) \subseteq \Sigma \times V$
Set of reactions contained in r	$\mathcal{N}(r) \subseteq \{n \in N \mid C(n) = r\}$
Set of mutations contained in r	$\mathcal{M}(r) \subseteq \mathcal{N}(r)$
Set of contained reactors of r	$\mathcal{R}(r) \subseteq \{r' \in R \mid C(r') = r\}$
Priority function	$\mathcal{P}(r) : \mathcal{N}(r) \to \mathbb{P}$
Startup trigger for r	$\bullet(r)$
Shutdown trigger for r	$\diamond(r)$
Reactor containing reactor r	$C(r) \subseteq R$

Table 2.2: A formal model of reactors.

share a particular port p among their effects, those reactions must be strictly well-ordered in $\mathcal{P}$, i.e., $\forall n, n' \in D^{\vee}(p)\ .\ C(n) = C(n') \wedge n \neq n' \implies \mathcal{P}(n) < \mathcal{P}(n') \vee \mathcal{P}(n') < \mathcal{P}(n)$. It should be emphasized that this choice was made merely to simplify the formalization. Any concrete runtime implementation could avoid this level of indirection and set the value of downstream ports directly.

Finally, each reactor has two reserved triggers: $\bullet$ and $\diamond$ which are used to signal the starting up or shutting down of the reactor, respectively. While these can be used to trigger reactions, the reaction code cannot schedule them. They can be seen as "hooks" for executing code at the beginning or end of a reactor's life cycle.

Ports

Definition 7 (Port)**.** *A **port** is defined as $p = (p, v)$, where*

1. *$p \in \Sigma$ is the **port identifier**; and*
2. *$v \in V$ is the **port value**;*

We will find it convenient to have auxiliary functions that return the reactions which have a given port as one of their sources (which include triggers), and the reactions which have a given port as their effect. To this end, we define the maps $\mathcal{N}(i)$ and $\mathcal{N}^{\vee}(p)$, respectively. Their definition is shown in Table 2.3.

Ports	
Port instance	$p = (x, v) \in P$
Port identifier	$x \in \Sigma$
Port value	$v \in V$
Reactions with p as a source	$\mathcal{N}(p) = \left\{ n \in \left(\left(\bigcup_{r \in \mathcal{R}(C(p))} \mathcal{N}(r) \right) \cup \mathcal{N}(C(p)) \right) \;\middle|\; p \in D(n) \right\}$
Reactions with p as an effect	$\mathcal{N}^{\vee}(p) = \left\{ n \in \left(\left(\bigcup_{r \in \mathcal{R}(C(p))} \mathcal{N}(r) \right) \cup \mathcal{N}(C(p)) \right) \;\middle|\; p \in D^{\vee}(n) \right\}$
Reactor containing p	$C(p) \subseteq R$

Table 2.3: A formal model of ports.

Actions

Definition 8 (Action)**.** *An **action** is defined as $a = (x, v, \mathfrak{o}, d, s, \mathfrak{p})$, where*

1. *$x \in \Sigma$ is the **action identifier**;*
2. *$v \in V$ is the **action value**;*
3. *$\mathfrak{o} \in \mathfrak{O}$ is the **action origin**, which specifies whether events on this action are to be scheduled relative to logical time or relative to physical time;*
4. *$d \in \{t \in \mathbb{T} \mid t \geq 0\}$ is the **minimum delay** of an event scheduled on this action with respect to the last-processed tag;*
5. *$s \in \{t \in \mathbb{T} \mid t \geq 0\} \cup \bot$ is the **minimum spacing** between any two events that are subsequently scheduled on this action, or $\bot$ if no constraint applies; and*
6. *$\mathfrak{p} \in \mathfrak{P}$ is the **spacing violation policy** of the action, which determines how violations of the minimum spacing requirement are handled.*

Action Origin

When an event is being scheduled on an action, this event will have a tag that is computed based on the minimum delay d, possibly an additional delay, and the current logical time or the current physical time, depending on whether the action's origin is *logical* or *physical*, respectively. An action a for which $\mathfrak{o}(a) = \text{Logical}$ is called a **logical action**; a **physical action**, on the other hand, is an action a for which $\mathfrak{o}(a) = \text{Logical}$.

Definition 9 (Origins)**.**

$$\mathfrak{O} = \{\text{Logical}, \text{Physical}\}$$

While this logic is detailed in Algorithm 2 in Section 2.5, the intent behind distinguishing logical and physical actions is that logical actions must only be scheduled during reactions, at well-defined logical time instants, whereas physical actions can be scheduled at any time, asynchronously (from another thread of execution), in response to something happening in the physical world. After all, there *is* no well-defined current logical time outside of the context of a reaction. Hence, for physical actions, not only the value of the event is an input to the system, but so is be the timestamp that will determine its tag.

The subtle interaction between logical and physical time in the reactor model can be understood as establishing an interface between inherently asynchronous and nondeterministic concurrent tasks on the one hand (e.g., a sensor that monitors a physical process) and deterministic computational tasks that benefit from testability and could require precise and predictable timing on the other (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 (and fundamentally compromising testability), reactors provide a model of computation that that reconciles these disparate views.

Event Spacing

In order to prevent overwhelming the runtime environment by inundating with events, actions can be parameterized with a constraint on the volume of events that can be scheduled per a given time interval. Having such constraint is critical for enabling schedulability analysis in the face of sporadic events.

Definition 10 (Minimum event spacing)**.** *We define the* ***minimum spacing*** *of an action a, denoted as $s(a)$, to be the non-negative minimum distance between the tags of any two subsequently scheduled events on a. If $s(a) = 0$ then the minimum distance is one microstep.*

If a minimum spacing has been specified (i.e., it is not $\perp$), then a *policy* determines how violations if the spacing requirement are handled. For instance, a broken sensor or unforeseen circumstance in the physical part of a cyber-physical system could cause a flood of events to be scheduled. The policy then specifies how the runtime system has to cope with this.

Definition 11 (Spacing violation policies)**.** *We define the* ***spacing violation policy*** *of an action a, $\mathfrak{p}(a) \in \mathfrak{P}$, where*

$$\mathfrak{P} = \{\text{Defer}, \text{Drop}, \text{Replace}\}.$$

The exact meaning of these policies is clarified in Algorithm 2, but they can be summarized as follows:

- Drop: Ignore the scheduling request;
- Replace: Attempt to update the previously scheduled event if it has not been handled yet; defer the event otherwise; and

- Defer: Schedule the event, but adjust its tag so that it satisfies the minimum spacing requirement.

Actions	
Action instance	$a = (x, v, \mathfrak{o}, d, s, \mathfrak{p}) \in A$
Action identifier	$x \in \Sigma$
Action value	$v \in V$
Action origin	$\mathfrak{o} \in \{\text{Logical}, \text{Physical}\}$
Minimum delay	$d \in \{t \in \mathbb{T} \mid t \geq 0\}$
Minimum spacing	$s \in \{t \in \mathbb{T} \mid t \geq 0\} \cup \bot$
Spacing violation policy	$\mathfrak{p} \in \{\text{Defer}, \text{Drop}, \text{Replace}\}$
Last scheduled event	$\mathcal{L}(a) \subseteq (\{a\} \times V \times \mathbb{G}) \cup \bot$
Reactor containing a	$C(a) \subseteq R$

Table 2.4: A formal model of actions.

In order to enforce a minimum spacing between scheduled events, some bookkeeping is required. For this, we introduce the following function that maps a given action to the last event that has been scheduled on that action, or $\bot$ if there is no such event:

$$\mathcal{L}(a) \subseteq (\{a\} \times V \times \mathbb{G}) \cup \bot.$$

Reactions

Definition 12 (Reaction)**.** *A **reaction** n is defined as $n = (D, \mathcal{T}, B, D^\vee, H, \Delta, B_\Delta)$, where*

1. *$D \subseteq I(C(n)) \cup \bigcup_{r \in \mathcal{R}(C(n))} O(r)$ is a set of **sources**, ports whose value the reaction may read;*
2. *$\mathcal{T} \subseteq D \cup \mathcal{A}(C(n)) \cup \{\bullet, \diamond\}$ is a set of **triggers**, whose presence cause the execution of the reaction;*
3. *B is the **body** of the reaction (i.e., the code that runs when the reaction executes);*
4. *$D^\vee \subseteq O(C(n)) \cup \bigcup_{r \in \mathcal{R}(C(n))} I(r)$ is the set of **effects**, ports whose value the reaction may write;*
5. *$H \subseteq \mathcal{A}(C(n))$ is the set of **schedulable actions**, actions for which n can generate events;*
6. *$\Delta \in \{t \in \mathbb{T} \mid t \geq 0\} \cup \bot$ is a **deadline** that, if not $\bot$, imposes a bound on the extent to which logical time is allowed to lag behind physical time when the reaction is triggered and ready to execute; and*

7. B_Δ *is the body of a* ***deadline miss handler****, which is an alternative reaction body to be executed when the deadline has been violated.*

Reactions	
Reaction instance	$n = (D, \mathcal{T}, B, D^\vee, H, \Delta, B_\Delta) \in N$
Set of reaction sources	$D(n) \subseteq I(C(n)) \cup \left(\bigcup_{r \in \mathcal{R}(C(n))} O(r)\right)$
Set of reaction triggers	$\mathcal{T}(n) \subseteq D(n) \cup \mathcal{A}(C(n)) \cup \{\bullet, \diamond\}$
Reaction body	$B(n)$
Set of reaction effects	$D^\vee(n) \subseteq O(C(n)) \cup \left(\bigcup_{r \in \mathcal{R}(C(n))} I(r)\right)$
Set of schedulable actions	$H(n) \subseteq \mathcal{A}(C(n))$
Reactor containing reaction n	$C(n) \subseteq R$
Reaction priority	$\mathcal{P}(n) \in \begin{cases} \mathbb{Z}^- & \text{if } n \in \mathcal{M}(C(n)) \\ \mathbb{Z}^+ \cup \{*\} & \text{otherwise} \end{cases}$
Priority of unordered reactions	$\forall p \in \mathbb{Z}^- \, \forall q \in \mathbb{Z}^+ . (p < *) \wedge (q \not< *) \wedge (* \not< q) \wedge (* \leq *)$
Deadline	$\Delta(n) \in \{t \in \mathbb{T} \mid t \geq 0\} \cup \bot$
Deadline miss handler	$B_\Delta(n)$

Table 2.5: A formal model of reactions.

Remark 4 (Reaction priority)**.** ***Reaction priority*** *determines the order in which reactions of the same reactor execute when triggered at the same logical time instant. $\mathcal{P}$ maps mutations to elements that are strictly less than the elements that it maps reactions to (see Table 2.5). Thus, a reactor's mutations will always have precedence over its reactions. The priority set includes a special priority element $*$ which is incomparable with the positive integers. It can be assigned to reactions that may execute in arbitrary order and therefore may execute concurrently, but only after all mutations of the reactor have finished executing.*

API for Reactions

A reactor program executes in the context of a runtime environment that provides the following procedures:

- currentTag: Returns tag $g = (t, m)$, the last observed logical time;
- get: Returns the value associated with given port/action at the current tag;
- physicalTime: Returns T, the last observed physical time;
- schedule: Schedules a given action with minimum delay of one *microstep*; and
- set: Binds a given value to a given port at the current tag;

- requestStop: To request the execution of the entire program to halt.

These procedures are the *only* means provided for code in the body of an ordinary reaction or deadline miss handler to interact with other reactors. While set and set facilitate synchronous communication with reactions in other reactors, schedule is intended to trigger reactions at a later tag within the *same* reactor, via an action. Actions can have a minimum 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 have an event occur with a tag relative to the last known logical time. On the other hand, actions with a physical origin (i.e., physical actions) allow events to be tagged based on a time value obtained from the platform (i.e., a physical clock).

Data Structures

Definition 13 (Event queue)**.** *We define the* ***event queue*** $\mathcal{Q}_E$ *as a set of scheduled events, to be handled no earlier than the moment at which physical time matches the time value of their tag.*

Definition 14 (Reaction queue)**.** *We define the* ***reaction queue*** $\mathcal{Q}_R$ *as a set of triggered reactions, to be executed in order of precedence at the current logical time.*

Definition 15 (Defunct reactor stack)**.** *We define the* ***defunct reactor stack*** $\mathcal{S}_D$ *as a set of defunct reactors (i.e, reactors that have been marked for deletion and are reacting to* $\diamond$ *at the current logical time), to be removed from their container after the last reaction at the current logical time has concluded.*

While we define $\mathcal{Q}_E$ and $\mathcal{Q}_R$ as ordinary sets in this formalization, any concrete implementation of a reactor runtime environment would use priority queues for these. Likewise, $\mathcal{S}_D$ is an ordinary set could conveniently be substituted with a stack to ensure that reactors are removed in the correct order (i.e., remove contained reactors before removing their container). While $\mathcal{Q}_E$ stores events that are to trigger reactions at some future instant, $\mathcal{Q}_R$ only stores reactions that have been triggered at the current logical time. Events are retrieved from $\mathcal{Q}_E$ ordered by tag (smallest tag first). Reactions are retrieved from $\mathcal{Q}_R$ ordered by precedence, which is determined through dependency analysis (see Section 2.6).

Event Generation

We now discuss how events are created. The body of a reaction is a container for application code. Let n be a reaction. Then the body $B(n)$ of this reaction is allowed to invoke the following two functions that affect the execution environment: schedule and set.

Setting Ports

A reaction can only execute SET on its declared effects. 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.

Algorithm 1 Set a value on port p and trigger reactions

```
1: procedure SET(p, value)
2:     v(p) ← value
3:     reactions ← T(p)
4:     LOCK(mutex)                    ▷ Obtain lock to ensure integrity of the reaction queue
5:     Q_R ← Q_R ∪ reactions          ▷ Queue triggered reactions for execution
6:     UNLOCK(mutex)                  ▷ Release the lock
7: end procedure
```

The name of this procedure was chosen carefully to reflect its semantics. Its invocation is analogous to the setting of a variable, not the sending of a message. Of subsequent invocations of SET on the same port at the same logical time only the last value will be observed by other reactors. Reactions triggered by the setting of a port are queued for execution at the same logical time instant.

Thread Synchronization

The procedures given as part of this formalization assume a multi-threaded execution platform. While it is not necessary for a reactor runtime environment to execute reactions in parallel, there are clear circumstances under which it could. The fact that opportunities for parallelization are statically known for a reactor program makes is a major attraction of the reactor model. While a non-threaded runtime environment would be somewhat simpler to explain, the complications introduced by threads are limited. Most importantly, thread synchronization is required to protect the integrity of concurrently accessed data structures. Specifically, $\mathcal{Q}_E$ and $\mathcal{Q}_R$, as well as the variable g that holds the current logical time, must be protected from data races. We achieve this using a *single* **mutex lock**, as shown in Algorithm 1, Lines 4–6. A major advantage of this design is that the use of a single lock ensures deadlock-freedom.

Threads do not only enable the parallel execution of independent reactions; they are also useful for facilitating non-blocking interactions with the (physical) environment—think of asynchronous callbacks or code in an interrupt service routine. Communication between such asynchronously executing code and the runtime system designed to occur through *physical* actions. Here, too, thread synchronization is necessary to protect the integrity of the event queue $\mathcal{Q}_E$ and the variable that stores the logical time g.

Scheduling Actions

A reaction can only call SCHEDULE on its set of schedulable actions. An invocation of SCHEDULE amounts to a request to have an event occur on a given action at some future instant. The logic used by the runtime environment to handle such request is shown in Algorithm 2, which is summarized as follows:

Algorithm 2 Schedule an action a.

```
1: procedure SCHEDULE(a, additionalDelay, value)
2:     delay ← d(a) + additionalDelay
3:     LOCK(mutex)                                   ▷ Ensure exclusive access to g and Q_E
4:     if 𝔬(a) = Logical then
5:         if delay = 0 then                         ▷ Compute tag for logical action
6:             tag ← CURRENTTAG()                    ▷ Preserve microsteps if delay is zero
7:         else
8:             tag ← (π1(CURRENTTAG()) + delay, 0)   ▷ Ignore microsteps otherwise
9:         end if
10:    else
11:        tag ← (PHYSICALTIME() + delay, 0)         ▷ Compute tag for physical action
12:    end if
13:    if s(a) = ⊥ then                              ▷ Defer to next available microstep
14:        conflicts ← {e' ∈ Q_E | a(e') = a ∧ π1(g(e')) = π1(tag)}
15:        tag ← max({tag} ∪ {g(e') | e' ∈ conflicts}) + (0, 1)
16:    else if L(a) ≠ ⊥ then                         ▷ Determine whether tag is "too early"
17:        if (tag < g(L(a)) + s(a)) ∨ (tag = g(L(a)) ∧ s(a) = 0) then
18:            if 𝔭(a) = Drop then
19:                UNLOCK(mutex)
20:                return                            ▷ Do not schedule
21:            else if 𝔭(a) = Replace ∧ L(a) ∈ Q_E then
22:                v(L(a)) ← value                   ▷ Update the value of the last event if still in Q_E
23:                UNLOCK(mutex)
24:                return
25:            else
26:                tag ← max(g(L(a)) + (s(a), 0), g(L(a)) + (0, 1))   ▷ Defer the event
27:            end if
28:        end if
29:    end if
30:    e ← (a, value, tag)
31:    Q_E ← Q_E ∪ {e}                               ▷ Enqueue the event
32:    L(a) ← e                                      ▷ Record e as the last event for a
33:    UNLOCK(mutex)
34: end procedure
```

- L2 Compute the scheduling delay by adding the minimum delay of the given action $d(a)$ and specified the additional delay.
- L4–12: Compute a preliminary tag. If the action's origin is Logical, then do this based on the current logical time g. Microsteps are only preserved if the computed scheduling delay is zero. If the action's origin is Physical, then compute the preliminary tag based on the current physical time.
- If no minimum spacing has been specified, then determine whether there are any events already queued for the same action that have a tag with a time value that matches the computed tag. We call these events *conflicting*.
  - L15: If there exist conflicting events, then adjust the timestamp of the computed tag to have a microstep larger than the greatest tag among the conflicting events. If there are no conflicts, just add one microstep to the computed tag.
- L16: If a minimum spacing has been specified, enforce it according to the specified policy. If no policy has been specified, the assumed policy is Defer. Enforcement is only necessary if a previous event has been scheduled on the same action with a tag that is closer to the computed tag than is permitted by the minimum spacing, or when the minimum spacing is zero and the tag of the last-scheduled event matches the time value of the computed tag.
  - L19 If the policy is Drop, simply drop the event and return without having inserted a new event into the event queue.
  - L22 If the policy is Replace and the previously scheduled event $\mathcal{L}(a)$ is still on the event queue, then update its value and return. No new event will be inserted in the event queue.
  - L26: If either the policy is Defer or it is Replace but the previously scheduled event $\mathcal{L}(a)$ has already left the event queue, then recompute the tag to satisfy the minimum spacing requirement. If the minimum spacing is greater than zero, then let the new tag be the tag of the previously scheduled event $g(\mathcal{L}(a))$ with the time value offset by the minimum spacing. If the minimum spacing is zero, then let the new tag be $g(\mathcal{L}(a))$ plus one microstep.
- L30–L32: Proceed to schedule the event by inserting it into the event queue.

Causality

Logical actions are always scheduled with a minimum delay of one microstep. A microstep delay is an increment of the index in superdense time [133, 151, 13] with respect to the current logical time. Actions therefore do not imply causal dependencies and cannot give rise to causality loops. As such, they are excluded from the dependency analysis described in Section 2.6.

Monotonicity

The *additional delay* parameter of SCHEDULE is useful for specifying "variable" delays, but it also enables non-monotonic scheduling behavior. That is, an event e can get scheduled *after* another event e' has already been inserted into the event queue and $g(e) < g(e')$. One practical implication of this sort of this behavior is that a computationally expensive search ($\mathcal{O}(|\mathcal{Q}_E|)$ for a min-heap) is necessary to determine whether conflicting events already exist in the event queue (see L14 in Algorithm 2). It should be noted that specifying a minimum spacing also forces monotonicity, reducing the detection of conflicts to an $\mathcal{O}(1)$ operation. The monotonicity of microsteps is enforced irrespective of the minimum spacing since microsteps cannot be controlled explicitly by the programmer (the delay parameter passed to SCHEDULE is a time value, not a tag).

Requesting Termination

Finally, reaction code can request termination of the entire program using the procedure REQUESTSTOP, shown in Algorithm 3, which only has the side effect of determining the logical time at which the runtime will perform its last series of reactions. When executing on a single machine, the delay between the tag of the request and the resulting g_{stop} is one microstep. The picture is inherently more complex in a distributed execution setting, which is why we leave δ_{stop} unspecified. To see how the runtime responds after g_{stop} has been set, refer to the DOSTEP procedure in Algorithm 12.

Algorithm 3 Request execution to come to a halt

```
1: procedure REQUESTSTOP()
2:     g_stop ← min(g_stop, g + δ_stop)        ▷ δ_stop is determined by the execution platform
3: end procedure
```

Mutations

Mutations are reactions that have the capability to structurally change a reactor (specifically: $\mathcal{R}$ and $\mathcal{N}$) during the course of execution. These changes can be carried out using the following API extension that is available *only* to mutations:

- CREATE: Creates a new reactor instance given a reference to a reactor class;
- DELETE: Deletes the reactor identified by a given references from its container;
- CONNECT: Connects the ports of two reactors; and
- DISCONNECT: Disconnects the ports of two reactors.

Creating New Reactors

A new reactor can be created using the CREATE procedure (see Algorithm 4), which creates an instance from a given class and adds it to the set of contained reactors of the instance serving as the container of the new instance. If CREATE is called at (t, m), then any reaction of the newly-created reactor and any reactor in its containment hierarchy that is triggered by $\bullet$ will execute at (t, m) also (see START in Algorithm 5), but not before the last mutation of the new reactor's container has finished executing (see Section 2.6).

Algorithm 4 Create a reactor instance given a reactor class and a container instance

```
1: procedure CREATE(class, container)
2:     r ← ν(class, c_inst)                          ▷ Get a new instance
3:     c_inst ← c_inst + 1                           ▷ Atomic update of the instantiation counter
4:     R(container) ← R(container) ∪ {r}             ▷ Add instance r to the container
5:     START(r)                                      ▷ Trigger startup reactions
6: end procedure
```

Algorithm 5 Start the execution of a reactor

```
1: procedure START(r)
2:     Q_R ← Q_R ∪ T(•(r))                           ▷ Stage for execution all reactions triggered by •(r)
3:     for each r' ∈ R(r) do
4:         START(r')
5:     end for
6: end procedure
```

Deleting Existing Reactors

A reactor can also be deleted at runtime. If DELETE (see Algorithm 6) is called at (t, m), then the given reactor is marked for deletion, any of its reactions triggered by $\diamond$ are queued for execution, and DELETE is called recursively on all reactors that the given reactor contains. All reactors deleted at (t, m) will have their **shutdown reactions** triggered at (t, m). When the runtime has finished executing all reactions triggered at (t, m), all reactors marked for deletion have any connections that they may still have removed, and the runtime frees any resources they might occupy (see Algorithm 12). When the next step is executed at (t', m') where $t' \geq t \wedge m > m'$, reactors that were deleted at (t, m) no longer exist.

Creating New Connections

Using CONNECT (see Algorithm 7), a mutation m can connect any ports that are in its sources $D(m)$ or effects $D^{\vee}(m)$, or ports of reactors that it created at runtime. Connection creation is, like reactor creation, logically instantaneous. All mutations of a container are

Algorithm 6 Delete a given reactor

procedure DELETE(r)
  $\mathcal{S}_D \leftarrow \mathcal{S}_D \cup \{r\}$ ▷ Mark r for removal at the end of the current step
  for each $r' \in \mathcal{R}(r)$ **do**
    DELETE(r')
  end for
  $\mathcal{Q}_R \leftarrow \mathcal{Q}_R \cup \mathcal{T}(\diamond(r))$ ▷ Stage for execution all of r's reactions triggered by $\diamond$
end procedure

Algorithm 7 Connect port p to downstream port p'

procedure CONNECT(p, p')
  if $p \in O(C(p)) \wedge p' \in I(C(p'))$ **then**
    $r \leftarrow C(C(p))$ ▷ Connect output to downstream input
  else
    if $p \in I(C(p)) \wedge p' \in I(C(p'))$ **then** ▷ Connect input to contained input
      $r \leftarrow C(p)$
    else if $p \in O(C(p)) \wedge p' \in O(C(p'))$ **then** ▷ Connect contained output to output
      $r \leftarrow C(p')$
    else
      error: *Cannot connect input to output.*
    end if
  end if
  if $D^{\vee}(p') \neq \emptyset$ **then** ▷ Check for conflicts
    error: *Connection would break strict total ordering among reactions that affect p'.*
  end if
  before $\leftarrow \delta_r$ ▷ Record causality interface δ_r before change
  $n \leftarrow (\{p\}, \{p\}, \{\text{SET}(p', \text{GET}(p))\}, \{p'\}, \emptyset, \bot, \{\})$
  $\mathcal{N}(r) \leftarrow \mathcal{N}(r) \cup \{n\}$; $\mathcal{P} \leftarrow \mathcal{P} \cup \{(n, *)\}$ ▷ Add relay reaction
  after $\leftarrow \delta_r$ ▷ Record causality interface δ_r after change
  if before $\neq$ after $\vee$ $isCyclic(\gamma_N(r))$ **then** ▷ Check for changed δ_r and causality loops
    $\mathcal{N}(r) \leftarrow \mathcal{N}(r) \setminus \{n\}$ ▷ Undo adding relay reaction
    error: *Connection would create direct feedthrough or causality loop.*
  end if
  if $v(p) \neq \varepsilon \wedge p \in I(r)$ **then**
    LOCK(mutex)
    $\mathcal{Q}_R \leftarrow \mathcal{Q}_R \cup \{n\}$ ▷ Propagate input to contained input
    UNLOCK(mutex)
  end if
end procedure

executed to completion before the start of any reaction of any contained reactor, including newly created reactors. Every contained reactor, including newly created reactors, will react to events that are present at the current logical time. If a connection is made between an upstream port that has a value, that value is also propagated to the downstream port that it is connected to.

We summarize CONNECT as follows.

- L2–7: Determine the containing reactor to add the connection to, based on whether this is an output-to-input, input-to-input or output-to-output connection.
- L10: Do not create input-to-output connections.
- L13–15: Do not proceed if the downstream port is already listed as an effect of an existing reaction.
- L16 Before enacting any changes, record the causality interface of the container.
- L17–L18: As explained in Remark 3, we use relay reactions to realize connections. We add the relay reaction to the appropriate container and assign it the priority $*$.
- L19 After making the change, again the record the causality interface of the container.
- L20–L23: Verify that the new connection has not altered the causality interface of the container or introduced cycles in the reaction graph of the container. If either of these things happened, remove the added relay reaction and report an error. For definitions of reaction graph $\gamma_N(r)$ and causality interface δ_r, see Section 2.6 (Definitions 16 and 18, respectively).
- L26: Finally, if the upstream port is an input of the containing reactor and it has a value, propagate the value to the downstream port.

Deleting Existing Connections

A mutation can also disconnect any two ports that it declares as a source or effect. It can also disconnect any ports of reactors that it has created at runtime (if it stored a reference to those instances). As with the creation of connections, these changes are reflected instantaneously, and are not witnessed by any contained reactors until they have been finalized for that time step. It is not necessary to remove connections to/from or inside a reactor that is deleted, as this will be taken care of automatically at the end of the time step at which the deletion happens. However, if the goal is to prevent certain reactions from occurring at the time of deletion, or to prevent any outputs created at time of a reactor's destruction from being witnessed by other reactors, one can remove connections manually to achieve this.

Of course, DISCONNECT essentially performs the inverse operation of CONNECT. For completeness, DISCONNECT is described in Algorithm 8.

Algorithm 8 Disconnect p from downstream port p'

```
1: procedure DISCONNECT(p, p′)
2:     if p ∈ O(C(p)) ∧ p′ ∈ I(C(p′)) then
3:         r ← C(C(p))                          ▷ Disconnect output from downstream input
4:     else
5:         if p ∈ I(C(p)) ∧ p′ ∈ I(C(p′)) then   ▷ Disconnect input from contained input
6:             r ← C(p)
7:         else if p ∈ O(C(p)) ∧ p′ ∈ O(C(p′)) then   ▷ Disconnect contained output from output
8:             r ← C(p′)
9:         else
10:            return
11:        end if
12:        n ← {n ∈ r | p ∈ D(n) ∧ B(n) = {SET(p′, GET(p))} ∧ p′ ∈ D^∨(n)}
13:        N(r) \ {n}; P ← P \ {(n, P(n))}                    ▷ Remove relay reaction
14:    end if
15:    if v(p) ≠ ε ∧ p ∈ I(r) then
16:        LOCK(mutex)
17:        Q_R ← Q_R \ {n}          ▷ Prevent propagation of input to contained input
18:        UNLOCK(mutex)
19:    end if
20: end procedure
```

2.6 Dependency Analysis

In the reactor model, each event has a tag. Reactions to events occur at a logical time equal to the tags of the events that are *present*, and logical time does not advance during a reaction. A port or action can have at most one event at any logical time. At any given logical time $g = (t, m)$, multiple reactions may be triggered. Some care is needed to ensure that triggered reactions execute in the correct order. Specifically, no reaction is to be executed before the values of all sources and reactor state that it depends have been determined. If during a reaction n triggered by an event with some tag g a particular trigger or source is *absent*, then it must be guaranteed that no event appears on that port or action with a tag equal to (or smaller than) g during or after the execution of n.

To determine the necessary constraints on the execution order of reactions, we first arrange reactions as vertices in a dependency graph using Algorithm 9, in which edges between reactions are implied by 1) the sources and effects of reactions; 2) priority with respect to other reactions within the same reactor; and 3) mutations. We can then use this graph decide whether a reaction r depends on another reaction r' and thus weather or it would be safe to execute r before or during r'. In a concrete implementation, one could assign each reaction an index based on a topological sort of the dependency graph and simply order reactions by index. This topic is discussed in more depth in Chapter 4.

Definition 16 (Reaction graph)**.** *Let r be a reactor. The* ***reaction graph*** *$\gamma_N(r)$ is a graph whose vertices are all reactions contained in the hierarchy of r and whose directed edges denote dependencies between vertices. It is computed according to Algorithm 9.*

Dependencies on Mutations

Before discussing Algorithm 9, let us consider one particular kind of dependency that must appear in the reaction graph; those between a reaction n and mutations that might affect the structure of its container $C(r)$. Specifically, we need to ensure that each *first* reaction (by priority) of a reactor depends on the *last* mutation (again, by priority) that exists up the hierarchy. Without such dependency, reactions could start executing while their upstream connections are being rerouted or their container is being deleted. We define an auxiliary function in order to find the nearest mutation:

$$mut(r) = \begin{cases} \{m \in \mathcal{M} \mid \forall m' \in \mathcal{M}(C(r)) \,.\, P(m') \leq P(m)\} & \text{if } C(r) \neq \top \wedge \mathcal{M}(C(r)) \neq \emptyset, \\ mut(C(C(r))) & \text{if } C(r) \neq \top \wedge C(C(r)) \neq \top \\ & \wedge \mathcal{M}(C(r)) = \emptyset, \\ \emptyset & \text{otherwise.} \end{cases}$$

Algorithm 9 Return the reaction graph of reactor r

1: **function** $\gamma_N(r)$
2: $(V, E) \leftarrow \bigcup_{r' \in \mathcal{R}(r)} \gamma_N(r')$ ▷ Contained reactors
3: $V \leftarrow V \cup \mathcal{N}(r)$ ▷ Reactions
4: $E \leftarrow E \cup \bigcup_{\substack{n \in \mathcal{N}(r) \\ p \in D(n)}} \{n\} \times \mathcal{N}^{\vee}(p)$ ▷ Sources
5: $E \leftarrow E \cup \bigcup_{\substack{n \in \mathcal{N}(r) \\ p \in D^{\vee}(n)}} \mathcal{N}(p) \times \{n\}$ ▷ Effects
6: $E \leftarrow E \cup \bigcup_{n,n' \in \mathcal{N}(r)} \{(n, n') \mid \mathcal{P}(n') < \mathcal{P}(n)\}$ ▷ Reaction priority
7: $E \leftarrow E \cup \{n \in \mathcal{N}(r) \mid \forall n' \in \mathcal{N}(r) \,.\, \mathcal{P}(n) = * \vee \mathcal{P}(n) \leq \mathcal{P}(n')\} \times mut(r)$ ▷ Mutations
8: **return** (V, E)
9: **end function**

The reaction graph is constructed as follows:

- L2. Make the vertices and edges of the dependency 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 reactions of r vertices of the graph.

- L4-5. Relate each reaction of r to other reactions based on its sources and effects. Note that the function $\mathcal{N}$, when applied to ports, returns the reactions that list the given port as a source; $\mathcal{N}^\vee$ returns the reactions that list the given port as an effect.
- L6. For all reactions of this reactor, add an edge to the graph between two reactions when the priority of one is smaller than the priority of the other (i.e., $C(n) = C(n') \wedge (\mathcal{P}(n) < \mathcal{P}(n'))$)
- L7. Make the first reaction of each contained reactor r' dependent on the last mutation that can affect the container r (if there is one). This mutation may be located further up the containment hierarchy. The dependency ensures that no contained reactions execute before mutations performed on the container are finalized.

After computing the dependency graph G using Algorithm 9, the graph must be checked for directed cycles. Cyclic dependency graphs must be rejected, as they represent causality loops; we do not handle them. If G is acyclic, then its reachability relation is a partial order [49]. It is this partial order that determines the execution order of reactions during the execution of a reactor.

We define $isCyclic : V \times (V \times V) \to \{true, false\}$ as

$$isCyclic(V, E) = \begin{cases} true & \text{if } \exists v \in V.(v, v) \in E^+, \\ false & \text{otherwise.} \end{cases}$$

Note that while actions may be featured in a reaction's sources and effects, they excluded from the dependency analysis because actions are always scheduled at least one microstep into the future.

Definition 17 (Reaction precedence)**.** *Given a top-level reactor r and its reaction graph $\gamma_N(r) = (V_R, E_R)$, a reaction n is said to be **dependent on** another reaction n', or equivalently, n' **precedes** n, if and only if n' is **reachable** from n, meaning that there exists a sequence of adjacent vertices in G_r (i.e., a path) which starts in r and ends in r'. In order to be able to test for the existence of a dependency between two reactions, we define the predicate $\prec_r : \mathcal{N} \times \mathcal{N} \to \{true, false\}$ which we define as follows:*

$$\prec_r (n', n) = \begin{cases} true & \text{if } (n, n') \in E_R^+ \\ false & \text{otherwise} \end{cases}$$

where E_R^+ denotes the transitive closure of E_R.

In a **dependency graph**, a directed edge denotes a "depends on" relation between two nodes. A dependency graph can also be encoded as a **precedence graph**, in which directed edges denote a "happens before" relation, a term that became famous due to Lamport's logical clock algorithm for achieving a causal ordering of events in a distributed system [118].

These representations are topologically identical, but the polarity of the edges is inverse. In the context of concurrency control in databases, precedence graphs are also commonly referred to as "conflict graphs" or "serializability graphs" [196]. In Definition 17, we chose to clarify what it means for there to exist a dependency between two reactions in terms of **precedence** because it aligns with the flow of information between reactions. Suppose we have two reactions n and n' where n depends on n' we can equally say that n' precedes n (i.e, $n' \prec n$) which is equivalent to say that n' is **upstream** relative to n, or, alternatively, n is **downstream** of n.

Similarly, we can construct a **port graph** for any given reactor—a dependency graph of which the vertices are ports (see Algorithm 10). This graph forms the basis of the definition of a reactor's causality interface (see Definition 18).

Algorithm 10 Report the dependencies between all ports in reactor r

1: **function** $\gamma_P(r)$
2: $\quad (V, E) \leftarrow \bigcup_{r' \in \mathcal{R}(r)} \gamma_P(r')$ ▷ Contained reactors
3: $\quad V \leftarrow V \cup I(r) \cup O(r)$ ▷ Ports
4: $\quad E \leftarrow E \cup \bigcup_{n \in \mathcal{N}(r)} D^{\vee}(n) \times D(n)$ ▷ Reactions
5: $\quad$ **return** (V, E)
6: **end function**

Definition 18 (Causality Interface)**.** *Given a top-level reactor r and its port graph $\gamma_P(r) = (V_P, E_P)$, a port p is said to be **dependent on** another port p' if and only if p' is **reachable** from p. In order to be able to test for the existence of a dependency between two ports, we use a **causality interface**, a predicate $\delta_r : I(r) \times O(r) \to \{true, false\}$ which we define (cf. [224]) as follows:*

$$\delta_r(n', n) = \begin{cases} true & \text{if } (n, n') \in E_P^+ \\ false & \text{otherwise} \end{cases}$$

where E_P^+ denotes the transitive closure of E_P.

Causality Loops

The dependencies imposed by reaction priority (i.e., the ordering of reactions within the same reactor) can have an unexpected side effect of introducing **causality loops**, which are cycles in the reaction graph. To explore this problem, let us examine a simple "rock, paper, scissors" game, illustrated in Figure 2.3. This simultaneous, two-player, zero-sum game, has only two possible outcomes: a draw, or a win for one player and a loss for the other. In case of a draw, the game repeats after approximately one second. The game works as follows. At the same logical time instant, each player picks either a rock, paper, or scissors,

and observes the other player's pick to determine the winner. A rock is defeated by paper. Paper is defeated by scissors. Scissors are defeated by a rock. The first reaction of each Player, triggered by either • or its logical action, randomly picks a symbol. The second reaction, triggered by the observe input, compares the two picks and either claims victory or scheduled the logical action in case of a draw. Naturally, each player has to pick a symbol *before* observing the other player's pick, or else they would be cheating. Interestingly, this kind of cheating is actually impossible using reactors.

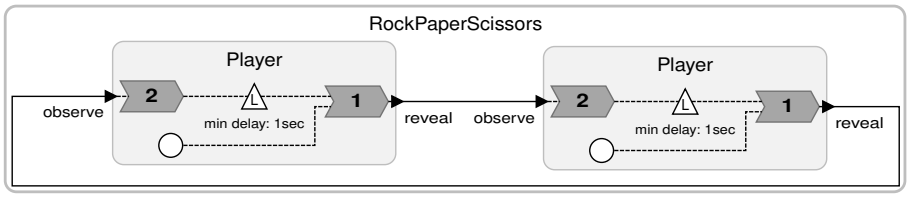


Figure 2.3: A reactor implementation of a simple "rock, paper, scissors" game.

If we were to swap the priorities of the two reactions, a causality loop would appear. While this might not come as a surprise in this particularly simple example, it is often more difficult to anticipate the emergence of causality loops in reactor programs. For this reason, a programming environment for reactors has to be equipped with a mechanism to accurately report these kinds of unanticipated causality loops. In our own IDE (see Section 3.1), we leverage our diagram synthesis tool to provide such feedback. The automatically synthesized diagram in Figure 2.4 highlights the direct feedthrough.

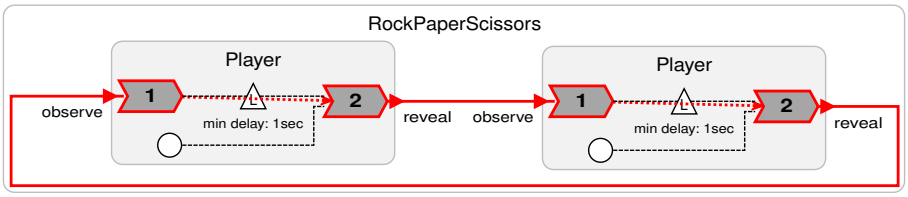


Figure 2.4: A causality loop due to reaction priority.

The rendering in Figure 2.5 goes a step further by filtering out all elements that are not part of the causality loop. As these diagrams are interactive, the programmer can simply click on the involved reactions to quickly navigate to their definition in the code.

2.7 Execution Algorithm

The execution of reactors in a **reactor runtime** is based on a discrete-event model of computation that guarantees determinacy, a property that can be proven by showing the existence of unique fixed points over generalized ultrametric spaces given that the reaction graph that governs the execution (see Definition 16 in Section 2.6) contains no directed cycles [140, 157].

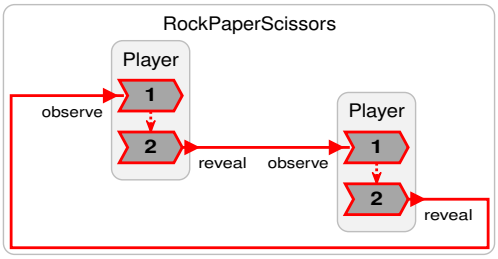


Figure 2.5: A filtered version of diagram in Figure 2.4.

The execution environment keeps a notion of a global event queue $\mathcal{Q}_E$ that tracks events scheduled to occur in the future, and a reaction queue $\mathcal{Q}_R$ for queueing reactions to be executed at the current logical time, in precedence order.

The execution of a reactor is captured in the EXECUTE procedure in Algorithm 11. At the beginning of execution, logical time starts at a value of $g = (T, 0)$, and it may increase as execution progresses. Logical time increases step-wise. A step starts when the previous step has concluded (if there is one) and there is an event in $\mathcal{Q}_E$ with a tag greater than or equal to the current physical time T. A step ends when $\mathcal{Q}_R$ is empty and all triggered reactions have finished executing. Execution can be subjected to a timeout by assigning a value g_{stop} that is offset with respect to $(T, 0)$. If no timeout applies, $g_{stop} = (\infty, 0)$.

Algorithm 11 Execute top-level reactor r

```
1: procedure EXECUTE(r)
2:     g = (T, 0)                    ▷ Set logical time equal to physical time
3:     START(r)                      ▷ Trigger startup reactions
4:     DOSTEP(r)                     ▷ Perform the first step
5:     while true do
6:         NEXT(r)                   ▷ Handle the subsequent events
7:     end while
8: end procedure
```

Start of Execution

All reactors have a special startup trigger •. It is present only at the logical time instant at which the reactor is created. It serves as a trigger for reactions that carry out initialization tasks. The invocation of START will push all reactions triggered by • onto the reaction queue $\mathcal{Q}_R$. Before entering the main loop, we call DOSTEP to perform the first step in which the queued reactions will get executed (as well as any subsequent reactions triggered in effect). The logic of DOSTEP procedure is described in Algorithm 12 and can be summarized as follows:

Algorithm 12 Execute triggered reactions until $\mathcal{Q}_R$ is empty

```
1: procedure doStep(r)
2:     if g = g_stop then
3:         shutdown(r)                                   ▷ Trigger all shutdown reactions
4:     end if
5:     repeat
6:         for all n ∈ execSet do
7:             if isDone(n) then                         ▷ Check whether executing element is done
8:                 doneSet ← doneSet ∪ {n}; execSet ← execSet \ {n}
9:             end if
10:        end for
11:        if Q_R ≠ ∅ then                               ▷ Execute something, if possible
12:            if threadIsAvailable() then
13:                P ← Q_R ∪ execSet
14:                readyForExec ← {p ∈ P | ∄p' ∈ P . p' ≺_r p)} \ execSet
15:                if readyForExec ≠ ∅ then
16:                    n ← Select(readyForExec); execSet, Q_R ← execSet ∪ {n}, Q_R \ {n}
17:                    if Δ(n) = ⊥ ∨ π_1(currentTag()) + Δ(n) < physicalTime() then
18:                        runInThread(n)
19:                    else
20:                        runInThread(B_Δ(n))
21:                    end if
22:                else
23:                    waitUntilNumberOfIdleThreadsHasIncreased()
24:                end if
25:            else
26:                waitUntilThreadHasBecomeAvailable()
27:            end if
28:        else
29:            if execSet ≠ ∅ then
30:                waitUntilNumberOfIdleThreadsHasIncreased()
31:            end if
32:        end if
33:    until Q_R ∪ execSet = ∅
34:    cleanup ()                                        ▷ Remove defunct reactors and dangling connections
35:    if g = g_stop then
36:        exit
37:    end if
38: end procedure
```

- L5-10. If a reaction that has been under execution is done, move that reaction to doneSet and remove it from execSet.
- L12–21. 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 L16, select one reaction from the set of ready-to-execute reactions. None of these reactions have any dependencies on other reactions that have been triggered but have yet to execute or finish executing at this logical time instant. For a definition of the predicate $\prec_r$, see Definition 17. If the selected reaction is *on time* (L18), then execute it in the available thread. If the reaction is *late* (L20), then execute the reaction's deadline miss handler instead.
- L23. 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`.
- L26. If there are pending tasks, but the runtime system does not have resources to accept a new task, then wait until it can accept a new task. Again, `pthread_cond_wait` could be used to implement the wait.
- L30. If there are no pending tasks, but there are tasks currently in execution, then wait until at least one of the tasks under execution finishes.
- L33. We iterate the loop L5-33 until there remain no reactions to be executed, and there are none currently under execution.
- L34: At the end of every step, remove all defunct reactors without leaving any dangling connections.

The CLEANUP procedure invoked on Line 34 of DOSTEP is shown in Algorithm 13. The order in which reactors are removed guarantees that no reactor gets removed before its contained reactors are removed first.

End of Execution

The logical time that marks the last step performed in the execution of a reactor is determined by g_{stop}, which is ∞ by default. This variable can either be set prior to execution as a means to enforce a *timeout*, or it can be set during execution through a call to REQUESTSTOP. Before DOSTEP starts executing reactions, it first checks whether the current logical time g is equal to g_{stop} (Lines 2–4) of DOSTEP, in which case it invokes SHUTDOWN which puts all reactions triggered by $\diamond$ on the reaction queue. If g_{stop} is found to be equal to the start time $(T, 0)$, then DOSTEP will execute exactly *once* and both $\bullet$ and $\diamond$ will be present for each reactor during this step. If $g = g_{stop}$, DOSTEP will exit the program rather than return (see Lines 35–37).

Algorithm 13 Detach and remove defunct reactors from reactor r

1: **procedure** CLEANUP(r)
2: **while** $\mathcal{S}_D \neq \emptyset$ **do**
3: $F \leftarrow \{r \in \mathcal{S}_D \mid \nexists r' \in \mathcal{S}_D \,.\, r' \in \mathcal{R}(r)\}$
4: $b \leftarrow \{\text{SET}(p', \text{GET}(p))\}$ ▷ Relay reaction body
5: $\mathfrak{D} \leftarrow \emptyset$ ▷ Build set of dangling connections
6: $\mathfrak{D} \leftarrow \mathfrak{D} \cup \{n \in \mathcal{N}(C(r)) \mid p \in I(C(r)) \wedge p' \in I(r) \wedge B(n) = b\}$
7: $\mathfrak{D} \leftarrow \mathfrak{D} \cup \{n \in \mathcal{N}(C(r)) \mid p \in O(r) \wedge p' \in O(C(r)) \wedge B(n) = b\}$
8: $\mathfrak{D} \leftarrow \mathfrak{D} \cup \left\{n \in \mathcal{N}(C(r)) \;\middle|\; p \in \left(\bigcup_{r \in \mathcal{R}(C(r))} O(r)\right) \wedge p' \in I(r) \wedge B(n) = b\right\}$
9: $\mathfrak{D} \leftarrow \mathfrak{D} \cup \left\{n \in \mathcal{N}(C(r)) \;\middle|\; p \in O(r) \wedge p' \in \left(\bigcup_{r \in \mathcal{R}(C(r))} I(r)\right) \wedge B(n) = b\right\}$
10: $\mathcal{N}(C(r)) \setminus \mathfrak{D}$ ▷ Remove dangling connections
11: **for all** $r \in F$ **do**
12: FREE(r); $\mathcal{S}_D \leftarrow \mathcal{S}_D \setminus \{r\}$
13: **end for**
14: **end while**
15: **end procedure**

Algorithm 14 Stop the execution of reactor r

1: **procedure** SHUTDOWN(r)
2: $\mathcal{Q}_R \leftarrow \mathcal{Q}_R \cup \mathcal{T}(\diamond(r))$ ▷ Enqueue all reactions triggered by $\diamond(r)$
3: **for each** $r' \in \mathcal{R}(r)$ **do**
4: SHUTDOWN(r')
5: **end for**
6: **end procedure**

Processing Events

Once the first invocation of DOSTEP has concluded and execution has not terminated, the main event loop is entered (Algorithm 11, Lines 5–7), which consists of repeatedly invoking NEXT. The NEXT procedure (Algorithm 15) is summarized as follows:

- L9-17. Determine what the next logical time should be, based on the event in $\mathcal{Q}_E$ that has the smallest tag, and wait for physical time to match its time value. The procedure TIMEDWAITFOREVENTQUEUECHANGE blocks until either the event queue was modified or the specified physical time was reached, whichever comes first. Upon being called, 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. In a single-threaded runtime a routine like `nanosleep` (POSIX) or `clock_nanosleep` (Linux) could be used. A bare-iron runtime will have to implement its own timer routine.

Algorithm 15 Process the next event(s) for a top-level reactor r

```
1: procedure NEXT(r)
2:     LOCK(mutex)                          ▷ Mutual exclusivity with concurrent SCHEDULE
3:     if (Q_E = ∅ ∧ ¬keepAlive) then
4:         g ← g + (0, 1)                                  ▷ Increment the microstep
5:         g_stop ← g                      ▷ End execution after completing a last step
6:         UNLOCK(mutex)
7:         DOSTEP(r)                                       ▷ Execute final step
8:     end if
9:     while true do
10:        T ← PHYSICALTIME()
11:        g_next ← min(g(PEEK(Q_E), g_stop))      ▷ Obtain the tag of the first-in-line event
12:        if (T, 0) ≥ g_next then
13:            break
14:        else                ▷ Wait until Q_E changes or physical time matches tag
15:            TIMEDWAITFOREVENTQUEUECHANGE(π_1(g_next))
16:        end if
17:    end while
18:    g ← g_next; Q_R, doneSet, execSet ← ∅, ∅, ∅                ▷ Advance logical time
19:    E ← {e ∈ Q_E | g(e) = g}; Q_E ← Q_E \ E      ▷ Gather events for current time t
20:    UNLOCK(mutex)                                              ▷ Release mutex
21:    CLEARALL()                                  ▷ Clear all inputs, outputs, actions
22:    for all e ∈ E do
23:        v(a(e)) ← v(e)                ▷ Set the value of the associated action a(e)
24:    end for
25:    Q_R ← ⋃_{e∈E} T(a(e))                  ▷ Enqueue reactions triggered by events
26:    DOSTEP(r)
27: end procedure
```

Algorithm 16 Recursively reset the values of all ports and actions of reactor r to absent

```
1: procedure CLEARALL(r)
2:     for all p ∈ I(r) ∪ O(r) do
3:         v(p) = ε
4:     end for
5:     for all a ∈ A(r) do
6:         v(p) = ε
7:     end for
8:     for all r' ∈ R(r) do
9:         CLEARALL(r')
10:    end for
11: end procedure
```

- L18. Advance logical time to match the smallest tag currently in $\mathcal{Q}_E$.
- L21. Set the values of all ports and actions to ε.
- L19. Obtain events to process at the current logical time.
- L20. Release the mutex, allowing concurrent calls to SCHEDULE to proceed.
- L22-24. Set triggers according to the value of the events pulled from $\mathcal{Q}_E$.
- L25. Obtain all reactions triggered by any of the events with a tag equal to the current logical time and insert them into $\mathcal{Q}_R$.
- L26: Perform another step.

The CLEARALL procedure invoked on Line 34 of NEXT is given in Algorithm 16. It simply ensures that any values that were set during a previous step are cleared.

2.8 Implementations

At the time of writing, there are several implementations of reactor runtimes in existence.

Reactor-C

Implemented by Edward A. Lee, Marten Lohstroh, and Soroush Bateni.

Because C is a rather low-level language, lacking a strong type system, memory management, and support for object-oriented design, it presents a number of challenges. On the other hand, C is the most universally supported language for embedded system design, and it runs efficiently on processors ranging from the smallest 8-bit microcontrollers to sophisticated 64-bit multi-core processors. A major goal of developing a **C target** is to quantify the minimal cost of supporting the deterministic concurrency model of reactors, a goal for which C is a suitable choice. Since the C runtime is designed purely as a target for code generation (not for standalone usage), it is developed as part of LF which we describe in Chapter 3.

Two implementations of the C runtime library exist. The first is suitable for very low-level embedded controllers, even those lacking an operating system. It relies on a subset of the C standard library. Any embedded platform with a C compiler and an implementation of this library can run the code generated by the LF compiler. For a bare-metal platform, we use `newlib`, a C standard library optimized for embedded systems. This implementation is suitable for embedded applications where most activities are periodic. The second implementation requires additionally a POSIX thread library. The addition of this library enables multi-core execution and integration of asynchronous external events (e.g., those generated by an interrupt request).

The C runtime provides utilities for dynamic memory allocation for non-primitive values (i.e., arrays and structs) created during reactions. After those values are passed along to other reactors, the burden of freeing the allocated memory is on the runtime. We use reference counting to determine when the memory occupied by such values can be freed.

The single-threaded C runtime consists of about 2,000 lines of extensively commented code; the threaded runtime has about 3,000 lines. A minimal application only occupies tens of kilobytes of memory, making it suitable for deeply embedded platforms. We have tested it on Linux, Windows, and Mac platforms, as well as on a bare-iron platform called Patmos [187]. On platforms that support pthreads (POSIX threads), it transparently exploits multiple cores while preserving determinism. A POSIX implementation for Patmos recently developed by Tórur Biskopstø Strøm has allowed us to successfully run our multi-threaded regression tests on Patmos as well. The runtime system includes features for real-time execution and is particularly well suited to take advantage of platforms with predictable execution times, such as Patmos and PRET machines [65, 225, 130].

Reactor-Cpp

Implemented by Christian Menard.

The C++ runtime is based on the **reactor-cpp** framework[2] which implements the reactor model. The framework provides mechanisms for specifying reactors and composing them, as well as the scheduler that is required for executing reactor programs. Similar to the C implementation, the scheduler transparently maps reactions to multiple threads for parallel execution while preserving determinism. The framework only depends on the standard template library (STL) of C++ and therefore executes on any platform that provides an STL implementation. It has also been shown that the framework integrates well with existing software frameworks. In particular, reactor-ccp has been used to augment the Adaptive Platform software stack that is part of the AUTOSAR automotive standard [162].

In contrast to C, C++ provides advanced support for object orientation, generic programming, and functional programming paradigms that allow for stricter enforcement of the reactor principles. Naturally, the concept of a reactor translates to the concept of a class in C++. In reactor-cpp, each reactor is represented as a specialized class that inherits basic reactor functionality from a common base class. The specialized class encapsulates all parameters, state, ports, actions, and reactions of the reactor while only exposing ports on its public interface. Ports and actions are implemented by generic classes and carry values of a fixed type. Only ports of the same type can be connected with each other. This enforces type-safety within the reactor network.

Each reactor program consists of multiple files: a header and a source file for each Reactor definition, a main file (`main.cc`) that controls the program execution, and a `CMakeLists.txt` file containing directives for an automatic build of the target. While the C++ code generator

[2] https://github.com/tud-ccc/reactor-cpp

enforces a more strict realization of the reactor principles than the C target, these principles can still be violated by reactions that are not well behaved. For instance, a reaction of reactor A could send a reference to the inner state of A to reactor B. Also, reactions of different reactors could have shared state, e.g., by using a common library that uses global variables. Strictly enforcing the reactor principles in C++ would only be possible by code inspection. Instead, reactor-cpp aims to prevent common mistakes and accidental violations of the reactor principles.

Another essential difference between the C and the C++ code generator is that the C++ implementation uses the ownership semantics of smart pointers to implement references to mutable and immutable values that are passed between reactors. When a reactor sends data to multiple downstream reactors, it is important that one of those reactors not be able to modify the data before it is seen by the other reactors. In this case, the downstream reactors will see immutable values. If one of those downstream reactors wishes to modify the data, it may request a unique pointer to a mutable version of the value. In most cases, this would create a copy of the original value and return a unique pointer to the mutable copy. If the reaction writes this mutable value to an output port, then the semantics of the unique pointer requires that the reaction transfers value ownership to the port. In other words, as soon as a mutable value leaves the scope of a single reaction, this reaction loses the capability to modify the value. In the special case where there is only one downstream reaction, the copying of the data can be avoided by passing a unique pointer in the first place, thereby enabling the downstream reactor to modify the data without compromising determinacy.

Reactor-TS

Implemented by Marten Lohstroh and Matt Weber.

Like reactor-ccp, the **reactor-ts**[3] framework that implements a TypeScript runtime was designed with standalone use in mind. Extra care has gone into ensuring type safety for programs written using reactor-ts. Reactions are modeled as instances of a `Reaction<T>` class where, type parameter `T` denotes the type of the argument list of the reaction function of type `(...args: ArgList<T>) => void` that implements the reaction body, which is passed into the constructor of `Reaction<T>` as an anonymous function. The type `ArgList<T>` is actually a *conditional type* [5] that is assignable only if all list elements are subtypes of `Variable`, an interface shared by all ports, actions, state variables, and parameters. The assigned type will be an inferred tuple type (essentially a list of types corresponding to the individual arguments), or, if the arguments have among them elements that do not subclass `Variable`, the assigned type will be `never`. We make use of TypeScript's `strictBindCallApply` compiler option to ensure that the actual argument list, which is also passed into the constructor of `Reaction<T>` matches the type signature of the reaction function that it is applied to.

[3] https://ts.lf-lang.org/

The runtime implementation depends on Node.js [207]. Of course, Node.js has its own event loop implementation, and does not provide access to threads. Because of this, the logic described in Algorithm 15 does not apply. Specifically, we must avoid blocking the event loop and use a timer to wake up when events in $\mathcal{Q}_E$ are due to be released. For this, we have a custom timer implementation that makes use of Node's `process.hrtime` with which we reach higher precision than is possible with the standard `setTimeout` routine.

Reactor-Py

Implemented by Soroush Bateni and Edward A. Lee.
The Python target developed for LF reuses the core of the C runtime through the use of Python C Extensions. It does not implement an independent runtime system.

> One thing [a language designer] should not do is to include untried ideas of [their] own. [Their] task is consolidation, not innovation.
>
> *C.A.R. Hoare*

Chapter 3

Lingua Franca

3.1 Overview

The goal of the reactor model is to provide effective means for building concurrent systems that can maintain an ongoing and interaction with their environment through a series of deterministic responses to external stimuli. The property of determinism makes rigorous testing much more feasible, but also allows for a more compositional approach to the design and implementation these kinds of systems. Indeed, the reactor model gives an unambiguous meaning to the *composition* of two reactors. But in order to be able to define reactors, specify their reactions, and compose them, we need a concrete software framework or programming language. To this end, we have developed Lingua Franca (**LF**).

As the name suggests, LF is intended as a "bridge language." One of the key advantages of reactors is that they can be coordinated as black boxes, without any knowledge about the specifics of the implementation of their reactions. LF capitalizes on this by concerning itself *only* with the definition and composition of reactors, and leaving the implementation of their reactions to some target language. LF is perhaps best described as a polyglot coordination language. The static semantics of an LF program can be understood in terms of mathematical objects discussed in Section 2.5; its dynamic semantics in terms of the algorithms that describe the reactor runtime system. We also discuss a formal semantics of LF based on fixed points in generalized ultrametric spaces in Section 3.8. LF currently supports C, C++, TypeScript, and Python. Rather than compete with immensely popular, feature-rich, and well-supported languages, LF is positioned to augment them with a deterministic coordination layer. This approach lets LF programs to leverage whatever libraries and compilers or interpreters the target language is equipped with.

Architecture

The general architecture of the Lingua Franca approach is outlined in Figure 3.1. An LF program, written in LF syntax, has to first be parsed and validated. Aside from straightforward syntax errors, there are less obvious problems that can render the program invalid.

For instance, the validator implements a static check for instantiation cycles (i.e., a reactor A instantiates reactor B which, in turn, instantiates A). It also makes sure that there exist no cyclic dependencies between reactions (i.e., the reaction graph as constructed in Algorithm 9 is acyclic). These are semantic checks that need to be performed to ensure that constructiveness of the program (see Section 3.8). It is also checked that all references to parameters, ports, and actions, are resolvable given the scoping rules enforced by LF. As a byproduct of the validation process, there is also the option of rendering the program graphically as an interactive diagram. These diagrams have proven to be very helpful for explaining the structure of the program. They also play a key role in error reporting as they can highlight the cause of aforementioned cycles, which are at times difficult to glean from the source code (see Section 2.6).

If no structural problems exist in the program, then the next step is to transpile the LF code to target code. While the structure of the resulting program is determined by the LF code, the implementation of reactions is given is verbatim target code, which is spliced directly into the generated code. In order to yield an executable program, the generated code has to be combined with a runtime implementation that is capable of coordinating the execution of reactors. If the target is a compiled language, the generated code has to first run through the target compiler, which is another point in the process where errors might occur. In order to relate compilation problems back to particular locations in the LF code, target-specific means are leveraged (e.g., the `#line` directive in C).

As a coordination language, LF governs the interactions and concurrent execution of chunks of target code. We make no attempt to limit what those chunks of code can do, and instead assume that they conform to the principles of reactors. The extent to which these principles can be enforced (such one reactor not sharing state with another reactor) varies between target languages. For example, in C there is little that can be enforced, and enforcement would likely add significant overhead. Hence, we assume that the chunks of target code are well-behaved. Better safety properties could be achieved by either code generating the chunks of C code from a safer language or using LF with a different target language, such as Java or Rust. Our C++ implementation, discussed in Section 2.8, already puts in place some guardrails to prevent target code from violating the reactor semantics.

Development Environment

Lingua Franca comes with a standalone command-line compiler called `lfc` and an Eclipse[1]-based IDE. The backbone of the compiler is the language implementation built using the Xtext [68] framework. Xtext applies a model-based approach to create an abstract syntax tree (AST) for a program. The grammar for a language is defined in extended Backus-Naur form from which Xtext derives a meta-model in the Eclipse Modeling Framework [200]. Xtext then provides extensions to populate data structures and set up cross-references in parsed AST models, for example, between the usage of a variable and its declaration. The

[1] https://www.eclipse.org/

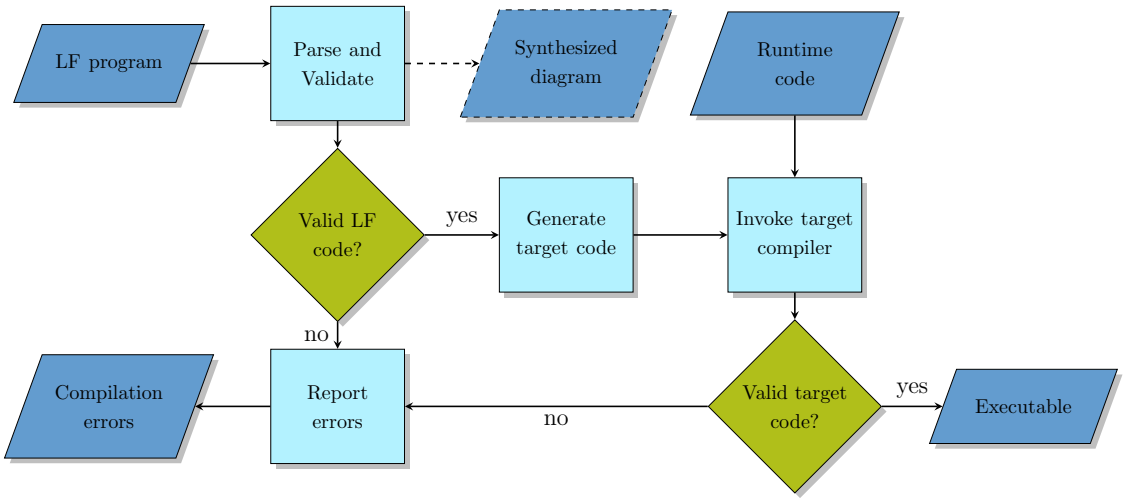


Figure 3.1: A flow chart describing the LINGUA FRANCA compiler toolchain.

parser for a language is generated from the grammar, as well as skeleton code for handling scoping, performing code validation (semantic checks on the AST), and code generation.

Furthermore, Xtext can automatically create editor support for syntax highlighting, content-assist, folding, jump-to-declaration, and reverse-reference lookup across multiple files. It can do this for the Eclipse-based editor, but (some of) these features are also available through a language server, extending the support to any editor that implements the language server protocol. This includes popular ones like VIM, Emacs, and Visual Studio Code. At the moment, a standalone Eclipse IDE for LF development with editor and compiler support is available and can be built and run from the LF repository.[2] In the future, we plan to additionally distribute a pre-built Eclipse application, editor plugins for integration into existing Eclipse installations, and a language server. Our goal is for LINGUA FRANCA IDE functionality to be easily integrated into existing development setups of (future) LF users.

Diagram Synthesis

The Eclipse-based LINGUA FRANCA IDE also provides automatically synthesized diagram representations for LF programs. They are based on the idea of **transient views** [186] that are created on-demand and usually focus on certain aspects of the program. This fits especially well with textual languages, such as LF, where a diagram allows for a fast and intuitive understanding of the general structure and important aspects of a program while the textual representation enables comfortable editing of every detail. A key enabler of this approach is the **automatic layout** [190]. It removes the tedious task of manually arranging elements in a diagram, which is especially undesirable when you are not even

[2] https://repo.lf-lang.org/

editing graphically in the first place. We implemented the diagram synthesis[3] for LF with the KIELER[4] Lightweight Diagrams framework [186]. Automatically generated diagrams are used throughout this thesis for illustrational purposes.

Syntactic Quirks in LF

Semicolons

Semicolons are *optional* in LF, but we do not feature them in our grammar definitions to avoid clutter. While some programmers may be used to write code without semicolons, others might have the habit to end every statement with a semicolon. While they are not technically necessary, it fits the spirit of a true "lingua franca" to allow both styles and provide a more fluid experience for the programmer who writes target code in a language like C that requires a semicolon at the end of each statement.

Comments

The LF syntax permits C/C++/Java-style comments and/or Python-style comments. All of the following are valid comments:

Listing 3.1: Using comments

```
// Single-line C-style comment.
/*
    Multi-line C-style comment.
*/
# Single-line Python-style comment.
'''
    Multi-line Python-style comment.
'''
```

Target Code

Verbatim target code can appear in several places in LF programs, such as in types and the body of a reaction. The {= =} delimiters are used to demarcate where target begins and ends. These **target code delimiters** consist of character sequences obscure enough that we have yet to encounter them in actual target code. As such, we have not yet seen the need to introduce an escape mechanism.

The `time` Type

LF generally does not do any type checking. If the target language has a static type checker, then the target compiler will fulfill this role. The common denominator among all target

[3] The diagram synthesis capability was contributed by Alexander Schulz-Rosengarten.

[4] https://rtsys.informatik.uni-kiel.de/kieler

languages we have considered thus far, however, is that they all lack a type for time values. Hence, LF fills in this gap with a **`time`** type. Time values are used in LF to specify behavior, and in order to be able to interpret such them correctly, they need to be accompanied by units. The LF validator checks this and reports it when units are missing, except when the value is zero. A time value has the following syntax:

$$
\begin{aligned}
\langle time \rangle &::= \langle INT \rangle \langle unit \rangle \mid 0 \\
\langle unit \rangle &::= \langle ns \rangle \mid \langle us \rangle \mid \langle ms \rangle \mid \langle s \rangle \mid \langle m \rangle \mid \langle h \rangle \mid \langle w \rangle \\
\langle ns \rangle &::= \texttt{`nsec'} \mid \texttt{`nsecs'} \\
\langle us \rangle &::= \texttt{`usec'} \mid \texttt{`usecs'} \\
\langle ms \rangle &::= \texttt{`msec'} \mid \texttt{`msecs'} \\
\langle s \rangle &::= \texttt{`sec'} \mid \texttt{`secs'} \\
\langle m \rangle &::= \texttt{`min'} \mid \texttt{`mins'} \\
\langle h \rangle &::= \texttt{`hour'} \mid \texttt{`hours'} \\
\langle w \rangle &::= \texttt{`week'} \mid \texttt{`weeks'}
\end{aligned}
$$

Lists

Lingua Franca also provides a convenient syntax for initializing arrays and lists. If the type in the target language is an array, vector, or list of some sort, then its initial value can be given as a list of values. For example, in the C target, you can initialize an array parameter as follows:

Listing 3.2: Using LF lists

```
reactor Foo(my_array:int[](1, 2, 3)) {
    ...
}
```

Equivalently, one could use target code delimiters, but this looks less elegant.

Listing 3.3: Declaring a static type initializer in verbatim C

```
reactor Foo(my_array:int[]({={1, 2, 3}=})) {
    ...
}
```

3.2 Target Declaration

Each LF program has to specify a **`target`**, which clarifies as to how the contents of reaction bodies are to be interpreted. The syntax is as follows:

$$\begin{aligned} \langle target \rangle &::= \langle ID \rangle (\text{`\{'} \langle property \rangle^* \text{`\}'})? \\ \langle property \rangle &::= \langle ID \rangle \text{`:'} \langle value \rangle \end{aligned}$$

A target specification may have optional parameters, called **target properties**, the names and values of which depend on the specified target. The syntax for specifying target properties is a simplified YAML [19] format limited to key-value pairs. Target parameters that are supported by all target languages are:

- `compiler` : A string giving the name of the target language compiler to use.
- `fast` : A boolean which, if `true`, specifies to execute as fast as possible without waiting for physical time to match logical time.
- `files` : A list of files to be copied to the directory that contains the generated sources.
- `flags` : A string giving options to be passed to the target compiler.
- `keepalive` : A boolean value to indicate whether to keep executing even if the event queue is empty. It is particularly useful to set this to `true` the execution is driven by sporadic physical actions (i.e, events scheduled in response to sensor input or network packet). By default, a program will exit once there are no more events to process.
- `no-compile` : If `true`, then do not invoke a target language compiler.
- `timeout` : A time value (with units) specifying the logical stop time of execution.

Sidebar: Metasyntax Notation

We define the LF syntax using a notation similar to Extended Backus-Naur Form (EBNF) notation used in Xtext[a]. The '?' as used in the ⟨*target*⟩ production denotes "zero or one" repetitions. The '*' and '+' operators denote "zero or more" and "one or more" repetitions, respectively. The binary '&' operator in the grammar joins elements into an "unordered group," where elements can occur in any order but each element may only appear once.

Bracketed terms written in capitals such as the ⟨*ID*⟩ represent terminals that are captured using some regular expression that is omitted for brevity. For instance, ⟨*ID*⟩ is specified as: `'^'?('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'\_'|'0'..'9')*`

[a] https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html#syntax

Example The target statement:

Listing 3.4: Example target statement with target properties

```
1 target C {compiler: "cc", flags: "-O3", fast: true, timeout: 10 secs}
```

specifies to use compiler `cc` instead of the default `gcc`, to use optimization level 3, to execute as fast as possible, and to exit execution when logical execution time has reached 10 seconds. The timeout effectively specifies $g_{\text{stop}} = g_{\text{start}} + 10\,\text{s}$. During the very last execution step at g_{stop}, all shutdown reactions will be triggered in addition to the reactions triggered by pending events with tag g_{stop}. Events on the event queue with a tag greater than g_{stop} will be not be handled.

3.3 Import Statement

It is also possible to import reactor definitions from other files. The **`import`** statement has the form:

$$
\begin{aligned}
\langle \textit{import} \rangle &::= \texttt{`import'}\langle \textit{reactors} \rangle\texttt{`from'}\langle \textit{file} \rangle \\
\langle \textit{reactors} \rangle &::= \langle \textit{rename} \rangle(\texttt{`,'}\langle \textit{rename} \rangle)^* \\
\langle \textit{rename} \rangle &::= \langle \textit{ID} \rangle(\texttt{`as'}\langle \textit{ID} \rangle)? \\
\langle \textit{file} \rangle &::= \texttt{`"'}\langle \textit{STRING} \rangle\texttt{`"'}
\end{aligned}
$$

where $\langle \textit{file} \rangle$ specifies another LF file in the search path, which currently only includes the location of the current file, but could be expanded in the future to include other locations, such as those listed in a project or package manifest. LF does not have a package system yet.

Example The following statement:

Listing 3.5: Example import statement

```
1 import Foo, Bar as Baz from "foobar.lf"
```

imports reactors `Foo` and `Bar` from "Foobar.lf", but it renames `Bar` to `Baz`, presumably to avoid a name collision with a local reactor named `Bar`. The renaming mechanism of imports obviates the need for so-called "fully qualified names" for disambiguation.

3.4 Preamble Block

Reactions may contain arbitrary target code, but often it is convenient for that code to invoke external libraries or to share function definitions. For either purpose, a reactor may

include a **preamble**. Notice that ⟨*code*⟩ can be *anything*, as long as it is delimited by an opening {= and closing =} (and does not include occurrences of either delimiter).

$$\begin{aligned} \langle \textit{preamble} \rangle &::= \langle \textit{visibility} \rangle ?\text{`preamble'} \langle \textit{code} \rangle \\ \langle \textit{code} \rangle &::= \text{`\{='.*`=\}'} \\ \langle \textit{visibility} \rangle &::= \text{`public'} \mid \text{`private'} \end{aligned}$$

The preamble can also be used to include local source files. In order for the files to be found, the files `files` target property can be used to automatically copy them into the output directory of the generated code.

The ⟨*visibility*⟩ modifier is not supported by all targets, hence it is not required. Its intent is to limit the scope in which the preamble code is visible. A preamble can occur in at the outer-most lexical scope of an LF file, or as part of the definition of a reactor. When a reactor is imported in another file, its private preamble is meant to *not* be visible in that file, whereas its public preamble *is*.

Example The following reactor uses the common `stdlib` C library to convert a string to an integer:

Listing 3.6: Using a preamble

```
1 target C;
2 main reactor StringToInt {
3     preamble {=
4         #include <stdlib.h>
5     =}
6     reaction(startup) {=
7         char* s = "42";
8         int i = atoi(s);
9         printf("Converted string %s to int %d.\n", s, i);
10    =}
11 }
```

When executed, this will print: `Converted string 42 to int 42.`

3.5 Reactor Definition

A reactor class definition is similar to a class definition in an object-oriented programming language like Java. It is structured as follows:

⟨*reactor*⟩ ::= ⟨*modifiers*⟩‘`reactor`’⟨*ID*⟩⟨*parameters*⟩⟨*host*⟩?⟨*parents*⟩?⟨*body*⟩
⟨*modifiers*⟩ ::= (‘`federated`’ | ‘`main`’)? &‘`realtime`’?
⟨*parameters*⟩ ::= (‘`(`’⟨*parameter*⟩(‘`,`’⟨*parameter*⟩)*‘`)`’)?
⟨*host*⟩ ::= ‘`at`’⟨*URI*⟩
⟨*body*⟩ ::= ‘`{`’(⟨*configuration*⟩ | ⟨*interface*⟩)*‘`}`’
⟨*configuration*⟩ ::= ⟨*preamble*⟩ | ⟨*var*⟩ | ⟨*instance*⟩ | ⟨*connection*⟩ | ⟨*reaction*⟩
⟨*interface*⟩ ::= ⟨*input*⟩ | ⟨*output*⟩ | ⟨*action*⟩ | ⟨*timer*⟩

Modifiers

Each file can only have only a single **`main`** keyword. It denotes the top-level reactor (see Definition 5 in Chapter 2) that is to execute when the program runs. When instead of **`main`** the **`federated`** keyword is used, this indicates that the program shall execute as a federation, where each reactor instance in the top-level reactor runs as a separate process, potentially at a distinct location. Federated execution is discussed in more detail in Section 5. That section also explains the purpose of the **`at`** clause in ⟨*host*⟩.

The **`realtime`** keyword is used to indicate so-called **realtime reactors**, which are committed to never run ahead of physical time because they interact with a sensor, actuator or some other source of sporadic events, such as a network interface. This is necessary to prevent a situation where a new event enters the system with a tag that is in the past relative to the current logical time. Any reactor that features physical actions is automatically considered a realtime reactor. The formalization of reactors in Section 2.5 makes no mention of this realtime property because the execution algorithm is conservative and considers *all* reactors to be realtime. Relaxation of the constraint that no event is handled before its tag has been surpassed by physical time, however, offers more scheduling flexibility and could increase the amount of exploitable parallelism during execution.

Parameters

Reactors can be given **parameters**, which can be overridden during initialization but are immutable after their initial assignment. The syntax for a parameter declaration is as follows:

⟨*parameter*⟩ ::= ⟨*ID*⟩‘`:`’⟨*TYPE*⟩?⟨*initializer*⟩
⟨*initializer*⟩ ::= ‘`(`’⟨*value*⟩(‘`,`’⟨*value*⟩)*‘`)`’
⟨*value*⟩ ::= ⟨*ID*⟩ | ⟨*time*⟩ | ⟨*literal*⟩ | ⟨*code*⟩
⟨*literal*⟩ ::= ⟨*INT*⟩ | ⟨*STRING*⟩ | ⟨*FLOAT*⟩ | ⟨*BOOL*⟩

Depending on whether the target language is statically typed, a type must be declared. For ⟨*TYPE*⟩ we allow common expressions for types, such as identifiers, rectangular brackets for arrays, stars for pointers, etc., but it is always possible to use ⟨*code*⟩ if the required syntax is not natively supported by LF.

Reactors do not have constructors, so each parameter has to be initialized with a default value. LF has native support for static list initializers as they are common in target languages; an ⟨*initializer*⟩ can either be a singleton value or a list of values. A ⟨*value*⟩ can either be a ⟨*time*⟩, ⟨*code*⟩, or an ordinary ⟨*literal*⟩. Because reactor definitions are not allowed to be nested, there are no parameters in scope that a ⟨*ID*⟩ can point to. But, as we will see, during instantiation of the reactor a parameter can be overridden with a value that refers to a parameter of the containing reactor.

Reactor Instantiation

Unlike objects classes, reactors classes do not have a constructor. Therefore, all contained reactor instances (analogue to class members in object orientation) must be statically initialized as part of their declaration. An instantiation looks as follows:

⟨*instance*⟩ ::= ⟨*ID*⟩`=` `new` ⟨*width*⟩? ⟨*ID*⟩ ⟨*assignments*⟩ ⟨*host*⟩?

⟨*assignments*⟩ ::= `(` ⟨*assignment*⟩ (`,` ⟨*assignment*⟩)*)? `)`

⟨*assignment*⟩ ::= ⟨*ID*⟩`=` ⟨*value*⟩

This syntax features the familiar **`new`** keyword commonly used in most object-oriented languages, and the kind of optional that one might recognize from class instantiation in Python. The optional ⟨*width*⟩ relates to LF's capability to create multiple instances at once. This feature is explained in more detail in Section 3.7. The optional ⟨*host*⟩ declaration can only be used for instances that part of the definition of a federated reactor, a concept explained in Chapter 5.

Example The following example prints `Hello World!` because `"Stranger"`, the default value of parameter `who`, is overridden with the string `"World"`.

Listing 3.7: Example of instantiation and parameter overriding

```
1 target TypeScript;
2 main reactor HelloWorld {
3     print = new Hello(who="World");
4 }
5 reactor Hello(who:string("Stranger")) {
6     reaction(startup) {=
7         console.log("Hello " + who + "!")
8     =}
9 }
```

Reactor instantiation gives rise to a containment hierarchy that is perhaps most apparent when an LF program is rendered as a diagram, as shown in Figure 3.2. Each instance is placed within a container instance, represented by a box with rounded corners. The outermost box denotes the `main` reactor. In the IDE, each reactor can be expanded or collapsed by double-clicking on it to show or hide its contents. The shown figure is fully expanded. The innermost reactor, which corresponds to the `Hello` reactor in Listing 3.7 only shows one reaction that is triggered by `startup`, which is represented by a circle (echoing the • notation used in Section 2.5).

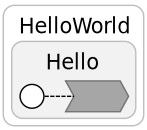


Figure 3.2: Graphical rendering of the "Hello World" program in Figure 3.7.

Inheritance

Reactors have their own **inheritance** mechanism that is fully independent of any inheritance mechanisms that may be featured in the target language. A reactor definition can declare which other reactors it extends using the following syntax:

$$\langle parents \rangle \quad ::= \quad \texttt{`extends'} \langle ID \rangle (\texttt{`,'} \langle ID \rangle)^*$$

All the ⟨*parameters*⟩, ⟨*configuration*⟩, and ⟨*interface*⟩ AST nodes are inherited from a reactor that gets extended. A reactor can only be extended successfully if all locally declared and inherited class members are of the same sort (e.g., if the extension has an ⟨*input*⟩ and any of the ⟨*parents*⟩ has an identically named member, it has to also be an ⟨*input*⟩) and their ⟨*TYPE*⟩ has to match. A reactor can extend multiple reactor classes. Special attention must be paid to the order in which they are listed in the ⟨*parents*⟩ clause; reactions, which are unnamed but *ordered* entities (see Remark 4 in Chapter 2) are inherited in the same order as their containing reactors appear in ⟨*parents*⟩.

Example Consider the code in Listing 3.8. The main reactor `SubclassesAndStartup` has an instance of `SubA` and an instance of `SubB`. Both `SubA` and `SubB` extend `Super`, so they each inherit the startup reaction from `Super`, and it has to execute before their own startup reaction. When executed, the output will show the output in Listing 3.9.

Note that when the target property `threads` is set to a value greater than `1`, then the order in which these print statements appear will permute across runs, but never will '`SubB started`' appear before '`SubB(Super) started`' or '`SubA started`' appear before '`SubA(Super) started`'.

Listing 3.8: Subclassing a reactor

```
target C;
reactor Super {
    reaction (startup) {=
        printf("%s(Super) started\n", self->name);
    =}
}
reactor SubA(name:string("SubA")) extends Super {
    reaction (startup) {=
        printf("%s started\n", self->name);
    =}
}
reactor SubB(name:string("SubB")) extends Super {
    reaction (startup) {=
        printf("%s started\n", self->name);
    =}
}
main reactor SubclassesAndStartup {
    a = new SubA();
    b = new SubB();
}
```

Output 3.9: SubclassesAndStartup

```
SubA(Super) started
SubB(Super) started
SubB started
SubA started
```

Ports

Port declarations have the form:

$$\begin{aligned} \langle input \rangle &::= \texttt{`mutable'}?\texttt{`input'}\langle width \rangle ?\langle ID \rangle \texttt{`:'}\langle TYPE \rangle ? \\ \langle output \rangle &::= \texttt{`output'}\langle width \rangle ?\langle ID \rangle \texttt{`:'}\langle TYPE \rangle ? \end{aligned}$$

The **mutable** keyword in $\langle input \rangle$ is a directive to the code generator indicating that reactions that read this input will also modify the value of the input. Without this modifier, inputs are considered *immutable*; modifying them is disallowed. The precise mechanism for making use of mutable inputs is target-language specific, and the extent to which immutability can be enforced varies from target to target.

The optional $\langle width \rangle$ relates to the notion of multiports that LF provides, which is described in more detail in Section 3.7. In a nutshell, this allows a single port to represent

not one, but multiple **channels**, each of which can receive data simultaneously from a different source. An ordinary port can only observe data coming from a single source.

Whether a ⟨*type*⟩ must be provided, is dependent on the target language. Unlike parameters, ports cannot be initialized with some value; they are always *absent* by default. If it is required, the LF validator will enforce it.

Actions

An action, like an input, can cause reactions to be invoked. Whereas inputs are provided by other reactors, actions are scheduled by this reactor itself, either in response to some observed external event or as a delayed response to some input event. The action can be scheduled by a reactor by invoking a SCHEDULE procedure provided by the runtime system.

An action declaration is structured as follows:

⟨*action*⟩ ::= ⟨*origin*⟩`'action'`⟨*ID*⟩⟨*timing*⟩?`':'`⟨*TYPE*⟩?
⟨*origin*⟩ ::= `'logical'` | `'physical'`
⟨*timing*⟩ ::= `'('`⟨*value*⟩(`','`⟨*value*⟩(`','`⟨*STRING*⟩)?)?`')'`

Like ports, actions carry values. If the target language is statically typed, then a ⟨*TYPE*⟩ must be provided.

Origin

An action always has to specify its ⟨*origin*⟩, which is either **`logical`** keyword or **`physical`** keyword. We refer to an action with a physical origin as a physical action and an action with a logical origin as a logical action. As laid out in Algorithm 2, the origin of the action passed to SCHEDULE determines whether the tag of the resulting event will be based on the time value of the current tag $\pi_1(g)$, or T, the current physical time. In either case, let us refer to this time value as the **time basis** for computing the tag of the resulting event. Additional parameters that affect the behavior of SCHEDULE are specified in the optional ⟨*timing*⟩ clause of the action. It is possible to specify ⟨*timing*⟩ parameters with references to parameters of the reactor (hence the use of ⟨*value*⟩ instead of ⟨*time*⟩). Of course, the LF validator will check whether parameters referenced in ⟨*timing*⟩ are of the **`time`** type and report an error if they are not.

Sidebar: Physical vs. Logical Actions

Physical actions are typically used to assign timestamps to external events, such as the arrival of a network message or the acquisition of sensor data, where the physical time at which these external events occurs is of interest. A typical use case for a physical action is to turn sensor readings into action events or to react to incoming messages received via a network. This can be achieved by invoking SCHEDULE in an asynchronous callback function or directly in an interrupt service routine (ISR).

Note that physical actions make it possible to inject into an executing program tagged events that result from asynchronous physical events outside the program. This goes considerably further than, for example, the timed extension of Esterel in [32], which provides mechanisms for controlling the timing of the execution of the program and hence for controlling the timing of its effects on the physical world. Physical actions enable more reactive programs; the program can react in predictable ways to unpredictable external events. As we will explain below in section 3.8, this may seem to undermine the determinacy of the LINGUA FRANCA, but if one considers the tag assigned to these external events as part of the *input* to the program, then the program remains deterministic. This is the key property that enhances testability; test vectors that include the timing of external events yield exactly one correct response.

Logical actions, on the other hand, can be used to achieve irregular (not periodic) events where the tag is under program control. Just like physical actions, logical actions can have an offset of zero. However, this does not result in an event at the same logical time that SCHEDULE is called because this could lead to nondeterminism. Instead, reactors adopt a *superdense* model of time [42, 151], in which each timestamp is replaced by a pair (t, m) that we call a tag, where t is a time value, and n a microstep index. This allows for the existence of events that have the same time value but are nonetheless ordered. Also see Section 5.1 for a discussion of our model of time. An event that was scheduled at tag (t, m) with zero delay will have a tag (t', m') such that $t' = t$ but $m' > m$.

Minimum Delay

The first $\langle value \rangle$ in $\langle timing \rangle$ denotes the minimum delay, which specifies the minimum distance between the tag of the resulting event and the time basis. It can be thought of as a minimum scheduling offset. Calling SCHEDULE on an action a with minimum delay $d(a)$ and additional delay d_{extra} communicates the intent of scheduling an event at $t_{\text{intended}} = B + d(a) + d_{\text{extra}}$, where B is the time basis, which is $\pi_1(g)$ if the origin $\mathfrak{o}(a) =$ Logical, and physical time T otherwise. This combination of a minimum delay, which is static and statically analyzable, and an additional delay, which can be computed at runtime, offers a balance that enables writing programs with strong guarantees that require static analysis, but also enables programs that require more flexibility.

Minimum Spacing

The second ⟨*value*⟩ in ⟨*timing*⟩ denotes the minimum spacing that has to be observed. It specifies the minimum distance between the tags of any two subsequently scheduled events on that same action. Setting a minimum spacing limits the extent to which invocations of SCHEDULE can overwhelm the runtime. It is particularly useful for physical actions, because the runtime cannot exercise any control over their scheduling. Without a minimum spacing requirement, physical actions would make schedulability analysis impossible. At runtime, spacing enforcement can also be used as a mechanism to implement back pressure such as is done in Reactive Streams [54].

If no minimum spacing is specified, then *no minimum spacing is enforced.*

Spacing Violation Policy

The third and last argument of ⟨*timing*⟩ is a ⟨*STRING*⟩ that denotes the spacing violation policy that shall be applied when the required minimum spacing between subsequently scheduled is violated. In other words, it determines is done when t_{intended} is *too close* to the tag of the last event that was scheduled on this action. The precise meaning of these policies is specified in Algorithm 2 in Section 2.5. We summarize them as follows:

- **"drop"**: Do not insert a new event;
- **"replace"**: Replace the last event; and
- **"defer" (default)**: Insert a new event, but adjust its timestamp so that the minimum spacing requirement is satisfied.

Timers

Timers are a convenient means for specifying periodic tasks, which are very common in embedded computing. For this reason, timers are offered as a language primitive in LF. The syntax of a timer declaration is as follows:

⟨*timer*⟩ ::= ⟨*ID*⟩(`'('`⟨*value*⟩(`','`⟨*value*⟩)?`')'`)?

A ⟨*value*⟩ in ⟨*timer*⟩ may again refer either to a parameter or a time value. The first ⟨*value*⟩ specifies the **offset**. The second ⟨*value*⟩ specifies the **period**. A timer triggers once at the start time plus the given offset, which is zero if it is left unspecified. If a period is given, then the action will continue to trigger repeatedly at regular intervals equal to the given period.

Listing 3.10: Using a timer

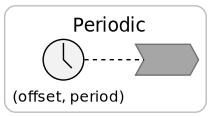
```
target C;
main reactor Periodic(
        offset:time(0),
        period:time(500 msec)) {
    timer t(offset, period);
    reaction(t) {=
        printf("%lld\n", get_elapsed_logical_time());
    =}
}
```

Listing 3.11: Using logical actions instead of a timer

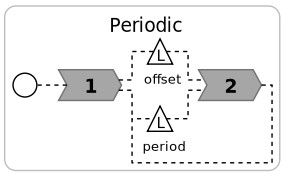
```
target C;
main reactor Periodic(
        offset:time(0),
        period:time(500 msec)) {
    logical action init(offset);
    logical action recur(period);

    reaction(startup) -> init, recur {=
        if (self->offset == 0) {
            printf("%lld\n", get_elapsed_logical_time());
            schedule(recur, 0, NULL);
        } else {
            schedule(init, 0, NULL);
        }
    =}

    reaction(init, recur) -> recur {=
    printf("%lld\n", get_elapsed_logical_time());
        schedule(recur, 0, NULL);
    =}
}
```

Example The timer on line 5 will trigger the reaction of the `Periodic` with a zero offset and period of 500 ms, which will print the elapsed logical time in nanoseconds (i.e., `0`, `500000000`, `1000000000`, ...). Timers are syntactic sugar. As shown in Listing 3.11, the exact same can be achieved using logical actions.

Diagrams of Listings 3.10 and 3.11 are shown in Figures 3.3a and 3.3a, respectively.



(a) Graphical rendering of Listing 3.10



(b) Graphical rendering of Listing 3.11

Figure 3.3: Timers are syntactic sugar for periodically recurring logical actions.

State Variables

A reactor may declare state variables, which are class members just like ports, timers, and actions; each reactor instance will have their own independent instances of these entities. State variables allow for reactions share information with other reactions within the same reactor and carry over information from one logical time to the next. A state variable is declared using the following syntax:

$$\langle var \rangle \quad ::= \quad \texttt{`state'}\langle ID \rangle \texttt{`:'}\langle TYPE \rangle ?\langle initializer \rangle ?$$

Example The following reactor will produce the output sequence 1, 2, 3, ... by incrementing the state variable `count` every time it reacts to timer `t`:

Listing 3.12: Using a state variable

```
1 target TypeScript;
2 main reactor Count {
3     state count:number(0);
4     timer t(0, 100 msec);
5     reaction(t) {=
6         count++;
7         console.log(count);
8     =}
9 }
```

While it may be tempting to use a **`preamble`** block for specifying shared state, doing so would lead to a very different behavior and, most likely, nondeterministic results. Unlike a state variable whose scope is limited to a single reactor instance, a global variable declared

in a in a preamble is shared among *all* reactor instances, which obviously leads to data races under multi-threaded execution. But even in purely sequential execution models, global variables have long been flagged as problematic. It was only a few years after Dijkstra's famous letter condemning the "goto" statement as harmful [60], that Wulf and Shaw argued for the abolishment of "non-local variables" [218]. The same reasons that applied then still apply now. Reactors formally do not feature global variables, and their usage in LF is strongly discouraged.

Connections

A reactor definition can specify connections between the ports of reactors it contains. It can also draw connections between the ports of contained reactors and its own ports. The syntax for specifying a connection is as follows:

$$\begin{aligned} \langle connection \rangle &::= (\langle ref \rangle \mid \langle refList \rangle)(\texttt{`->'} \mid \texttt{`\textasciitilde>'})(\langle ref \rangle \mid \langle refList \rangle) \\ \langle ref \rangle &::= \langle ID \rangle(\texttt{`.'}\langle ID \rangle)? \end{aligned}$$

The dotted notation in $\langle ref \rangle$ is used to refer to ports of contained reactors, where the first $\langle ID \rangle$ is the name of the port and the second $\langle ID \rangle$ refers to the name of the contained reactor. If no dot is used, then it is implied that the $\langle ID \rangle$ refers to a port of the containing reactor (i.e., a port of the reactor class in whose lexical scope the connection is defined). It is also reference multiple ports at once using a $\langle refList \rangle$, the details of which we discuss in Section 3.7.

Logical Connections versus Physical Connections

We distinguish two types of connections: **logical connections**, denoted by a straight arrow `->`, and **physical connections**, denoted by a squiggly arrow `~>`. Whereas the logical connection is synonymous with an ordinary connection in the reactor model (see Section 2.3), physical connections are syntactic sugar for a reaction that gets triggered by the upstream port, schedules a physical action, at some later tag gets triggered by the resulting event, and then sets the port of the downstream port. The net effect of this level of indirection is that data gets transferred between the upstream and downstream port without implying a dependency between those ports. This can be useful if no synchronization is necessary between the produced output and downstream reactions that observe it. One can compare this sort of interaction with the handling of events in JavaScript [149]; when an event occurs, it will be pushed onto a queue and handled later, in some arbitrary order. In other words, physical connections are a mechanism for intentionally introducing nondeterminism in places where no strict ordering of events is required.

Example We can show the difference in timing behavior between using a physical connection and using a logical connection using the code in Listing 3.13 by simply changing

Listing 3.13: Printing a timed sequence through a logical connection

```
1 target C {timeout: 599 msec};
2 main reactor TimedSequence {
3   ramp = new Ramp();
4   print = new Print();
5   ramp.y -> print.x;
6 }
7 reactor Ramp {
8     timer t(0, 100 msec);
9     output y:int;
10    state count:int(0);
11    reaction(t) -> y {=
12      SET(y, self->count);
13      self->count++;
14    =}
15 }
16 reactor Print {
17    input x:int;
18    reaction(x) {=
19        printf("Logical time: %lld, Physical time: %lld"
20            ", Value: %d\n",
21            get_elapsed_logical_time(),
22            get_elapsed_physical_time(), x->value);
23    =}
24 }
```

Output 3.14: `TimedSequence` with logical connection

```
Logical time: 0, Physical time 442124, Value: 0
Logical time: 100000000, Physical time 100146322, Value: 1
Logical time: 200000000, Physical time 200140962, Value: 2
Logical time: 300000000, Physical time 300146657, Value: 3
Logical time: 400000000, Physical time 400092148, Value: 4
Logical time: 500000000, Physical time 500142916, Value: 5
```

Line 5 from `ramp.y -> print.x` to `ramp.y ~> print.x`. When we compare the output in Output 3.14 to Output 3.15, we see that the logical times are no longer exact when the physical connection is used. Note that extra slack was added to the `timeout` target property to account for the time delay incurred by the physical connection (also see the discussion in Section 5.4 about coordinating the end of execution). The logical time of the last event in Output 3.15 is 500 153 069 ns (rather than 500 000 000 ns), meaning that only four events would show if the timeout was chosen to be exactly 500 ms.

Output 3.15: `TimedSequence` with physical connection

```
Logical time: 103524, Physical time 160233, Value: 0
Logical time: 100075065, Physical time 100130927, Value: 1
Logical time: 200130178, Physical time 200202198, Value: 2
Logical time: 300101395, Physical time 300168437, Value: 3
Logical time: 400025707, Physical time 400079239, Value: 4
Logical time: 500087864, Physical time 500153069, Value: 5
```

3.6 Reaction Definition

A reaction definition is somewhat similar to that of a class method or function. A reaction consists of ⟨*code*⟩, a block of code surrounded by target code delimiters, and a *signature*. Whereas the signature of a function usually consists of parameters and their types, a return type, and possibly extra annotations such as visibility modifiers and exception declarations, a reaction signature consists of a list of ⟨*triggers*⟩, ⟨*sources*⟩, and ⟨*effects*⟩, and an optional ⟨*deadline*⟩. The syntax is as follows:

⟨*reaction*⟩ ::= `'reaction'`⟨*triggers*⟩⟨*sources*⟩⟨*effects*⟩?⟨*code*⟩⟨*deadline*⟩
⟨*triggers*⟩ ::= `'('`⟨*io*⟩`')'`
⟨*sources*⟩ ::= ⟨*io*⟩?
⟨*effects*⟩ ::= `'->'`⟨*io*⟩
⟨*io*⟩ ::= ⟨*ref*⟩(`','`⟨*ref*⟩)*
⟨*deadline*⟩ ::= `'deadline''('`⟨*time*⟩`')'`⟨*code*⟩

An entry in ⟨*triggers*⟩, ⟨*sources*⟩[5], or ⟨*effects*⟩ may refer to a port or action of the containing reactor, or a port of a contained reactor. A reaction has to have at least one trigger (or else it would never execute). There are two additional triggers that may be used: **`startup`** for triggering the reaction at the very first logical time instant of the containing reactor's life cycle, and **`shutdown`** for triggering the reaction at the very last logical time instant of the container's existence. Ports or actions that should not trigger the reaction, but whose value might be read in the reaction's ⟨*code*⟩, must be listed among the reaction's ⟨*sources*⟩. Failure to do so will cause a compilation error. Ports whose value might be set in the reaction's ⟨*code*⟩ and actions that might be scheduled in the reaction's ⟨*code*⟩ must be listed among the ⟨*effects*⟩, i.e., after the `->`. Again, if a reaction sets a port or schedules an action that is not among its ⟨*effects*⟩, the compilation error will result.

While actions do not imply dependencies, making them part of the reaction signature allows their effects to be considered as part the static analysis of the program. For instance, if a reaction n schedules an action a that has a minimum delay $d(a)$, then a will not trigger

[5] In the formalization, sources include triggers, but in LF we separate them to avoid verbosity.

any reactions until $T > \pi_1(g) + d(a)$, in which T denotes physical time and g is the tag at which a was scheduled. Facts like these determine whether a program is **schedulable** or not (i.e., whether timing constraints can always be met) [159].

Timing constraints can be specified using a ⟨*deadline*⟩. A ⟨*deadline*⟩ is a property of a ⟨*reaction*⟩ that stipulates that whenever the reaction gets triggered, its ⟨*code*⟩ is to execute before $T > g + \Delta$, where T denotes the current physical time, g denotes the current logical time, and Δ is specified by the ⟨*time*⟩ parameter of the ⟨*deadline*⟩. Should a deadline miss occur at runtime, the ⟨*code*⟩ of the ⟨*deadline*⟩ will be invoked rather than the ⟨*code*⟩ of the ⟨*reaction*⟩. This behavior is codified in Algorithm 12 in Section 2.7.

Example The example discussed in Section 2.4 has an implementation that is available in the LF repository. Instead of interacting with physical pedals, brakes, and a motor, this demo receives input from a keyboard and prints a log of the performed control to `stdout`. The LF code for the Brake reactor is sketched in Listing 3.16. When the deadline is brought down sufficiently, say to 100 microseconds, deadline misses will start to occur. When this happens, the alternative reaction body on Line 8 is invoked instead of the reaction body on Line 6. While in a braking system even late application of the brakes is probably better than none at all, there exist scenarios in which late actuation can do substantial harm. For instance, imagine an automatic lane switching system that has to query a number of sensors and process their data in order to determine whether it is safe to switch lanes. Any conclusion that such system reaches that is not confined to a very limited time window would be dangerous to act upon.

Listing 3.16: Using a deadline

```
1 reactor Brakes {
2     input force:int;
3
4     // @label Reaction with deadline
5     reaction(force) {=
6         // On time
7     =} deadline (2 msec) {=
8         // Too late
9     =}
10 }
```

3.7 Banks and Multiports

In programs that require a "dense" connection topology, i.e., where some reactors have to interact with a significant number of peers, it can quickly become tedious to allocate ports, draw each individual connection, and specify reaction signatures. To avoid this, LF features ports that can send or receive over multiple channels, called **multiports**, and bundles of instances of a reactor class, called **banks of reactors**.

To declare an input or output port to be a multiport, or instantiate a bank of reactors instead of a single one, the optional **width** parameter must be used, which can be a positive integer constant ⟨*INT*⟩ or a parameter name ⟨*ID*⟩.

⟨*width*⟩ ::= '['⟨*INT*⟩ | ⟨*ID*⟩']'

Connecting Equal-width Multiports

Example Consider the code in Listing 3.17. This example features two multiports that are connected using an ordinary ⟨*connection*⟩ on Line 23, which connects each channel of `a.out` to `b.in`. In this case, the width of the ports match. If the widths do not match, then a warning is issued. The data of unconnected output channels gets discarded and unconnected input channels are always absent. This statement is legal because the widths of `a.out` and `b.in` match. The ⟨*width*⟩ of `a.out` is determined by the parameter `width`, which has a default value of `1` but is overridden during the instantiation of `a`, whereas `Destination.in` has a fixed width of 4. Downstream reactor `b` simply adds up the values 0, 1, 2, and 3 that it collects from the four channels of `b.in` and it prints:

```
Sum of received:  6.
```

A reaction triggered by a multiport will be triggered when *any* of the multiport's channels have a value. Hence, when using multiports, it is important to test for presence of the input on each channel, as done in Listing 3.17 on Line 15

When a ⟨*reaction*⟩ declares a multiport as a ⟨*trigger*⟩ (Line 12), ⟨*source*⟩, or ⟨*effect*⟩ (Line 4), then this brings into scope of the reaction body a way to access the width of the port and a way to write to each channel of the port. It is also possible to test whether a previous reaction has set an output value and to read what that value is (values are not actually cleared in between steps, only the indicator of their presence is reset). The exact syntax for this depends on the target language. In the C target, the width of a port name `out` is accessed with the variable `out_width`, and `x[i]` references the output channel to write to using the `SET` macro, as shown on Line 6.

Connecting Different-width Multiports

It is also possible to connect different-width ports using a ⟨*refList*⟩ on either side of the arrow in a ⟨*connection*⟩. The ⟨*refList*⟩ production is defined as follows:

⟨*refList*⟩ ::= '('⟨*ref*⟩(','⟨*ref*⟩)*')''+'?

Instead of referencing a single port, this construct allows one to reference lists of ports. The channels of ports on the left of the arrow are connected to channels of ports on the right, in the order they are listed (from left to right), until there are no more available channels on the right. If the total width on the left does not match the total width on the right, then a warning is issued. If the left side is wider than the right, then output data will be discarded. If the right side is wider than the left, then inputs channels will be absent.

Listing 3.17: Reactors with multiports

```
target C;
reactor Source(width:int(1)) {
    output[width] out:int;
    reaction(startup) -> out {=
        for(int i = 0; i < out_width; i++) {
            SET(out[i], i);
        }
    =}
}
reactor Sink(width:int(1)) {
    input[width] in:int;
    reaction(in) {=
        int sum = 0;
        for (int i = 0; i < in_width; i++) {
            if (in[i]->is_present) sum += in[i]->value;
        }
        printf("Sum of received: %d.\n", sum);
    =}
}
main reactor MultiportToMultiport {
    a = new Source(width = 4);
    b = new Sink(width = 4);
    a.out -> b.in;
}
```

Listing 3.18: Connecting multiports

```
target C;
import Source, Sink from 'MultiportToMultiport.lf' // See Listing 3.17.
main reactor MultiportToMultiport2 {
    a1 = new Source(width = 3);
    a2 = new Source(width = 2);
    b = new Sink(width = 5);
    a1.out, a2.out -> b.in;
}
```

Example Consider the code in Listing 3.18, which has a main reactor with two `Source` instances: `a1` that has `width=3` and `a2` that has `width=2`. It also has a `Sink` `b` with `width=5`. On Line 7, the first three channels of `b` are connected to the outputs of `a1` and the last two channels of `b` are connected to `a2`.

Listing 3.19: A multicast connection

```
1 target C;
2 import Source, Sink from 'MultiportToMultiport.lf' // See Listing 3.17.
3 main reactor Multicast(width:int(4)) {
4     a = new Source(width = 1);
5     d = new[width] Destination(width = 1);
6     (a.out)+ -> d.in;
7 }
```

Multicast Connections

It is also possible to have fewer ports on the left of a connection and have their channels multicast to ones on the right. To signal this intent, the optional parentheses and '+' in the ⟨*refList*⟩ must be used. The content inside the parentheses can be a comma-separated list of ports, the ports inside can be ordinary ports or multiports, and the ports be can members of ordinary reactors or banks of reactors. In all cases, the number of ports inside the parentheses on the left must divide the number of ports on the right.

Example The statement `(a.out)+` in Line 6 of Listing 3.19 means "repeat the output port `a.out` one or more times as needed to supply all the input ports of `d.in`."

Using Banks of Reactors

Like an ⟨*input*⟩ or ⟨*output*⟩, an ⟨*instance*⟩ can be parameterized with a ⟨*width*⟩. This syntax allows for the creation of a *bank* of reactors where ⟨*width*⟩ specifies the number of reactors in the bank. Banks and multiports can be combined; a ⟨*ref*⟩ in a ⟨*connection*⟩ can refer to a port in a single instance, to a multiport in a single instance, to a regular port in a bank of reactors, or to a multiport in a bank of reactors. Whenever the total number of channels on left side of a ⟨*connection*⟩ does not match the total number of channels on the right, a warning will be issued. To distinguish the instances in a bank of reactors, the reactor will automatically have a parameter called `bank_index` of type `int`. This will be assigned a number between 0 and $n - 1$, where n is the number of reactor instances in the bank.

Example The connection between `a.out` and `b.in` in Listing 3.20 is balanced.

Listing 3.20: Connecting banks of reactors

```
1 target C;
2 import Source, Sink from 'MultiportToMultiport.lf' // See Listing 3.17.
3 main reactor BankToBankMultiport {
4   a = new[3] Source(width = 4); b = new[4] Sink(width = 3);
5   a.out -> b.in; // 3 * 4 = 12 channels
6 }
```

Example In Listing 3.21, we connect a `Source` with a multiport of width 3 to a bank of three `Sink` reactors. Because each `Sink` instance has an input of width 1, the connection is balanced.

Listing 3.21: Connecting a multiport to a bank

```
1 target C;
2 import Source, Sink from 'MultiportToMultiport.lf' // See Listing 3.17.
3 main reactor MultiportToBank {
4     a = new Source(width=3); b = new[3] Sink();
5     a.out -> b.in;
6 }
```

3.8 Semantics

Lingua Franca, with its timestamped events, is rooted in a discrete-event model of computation. We can leverage prior work with the semantics of discrete-event systems [182, 220, 127, 222, 42, 139, 156, 158] to prove determinism. A program is deterministic if it exhibits exactly one behavior for each set of inputs. Some care is needed, however, because this statement requires defining precisely what we mean by "behavior" and "input."

First, LF cannot be fully dealt with by the DEVS formalism of [222] because there is no requirement for a nonzero logical time delay from inputs to outputs of reactors. Outputs are simultaneous (in logical time) with inputs, much like the synchronous languages [20].

Second, LF uses a superdense model of time [151, 155], where there is no requirement for a delta-causal component in feedback loops. As a consequence, the metric-space semantics of [182, 220, 127], which uses the Cantor metric and the Banach fixed point theorem, cannot be used unmodified. We can choose a semantics based on **complete partial orders (CPOs)** [139] or on a **generalized ultrametric space** [42, 156, 158]. Here we choose the latter. We will not give the full formalism here, since it is well documented in the literature, but instead will only explain how to map LF onto this formalism. A full understanding will require reading the prior work.

We use the concept of a **signal** to represent the sequence of timestamped messages that flow from output ports to input ports in LF. Formally, a signal is a partial function $s : T \rightharpoonup V$, where T is the *tag set* and V is the set of possible message values. A signal is *defined* for tags where there is an event (a message is sent) and is *undefined* for other tags. For the purposes of proving determinism, we take "behavior" to be the set of signals produced by a program execution.

The prior work with ultrametric space semantics assumes a superdense time tag set $T = \mathbb{R} \times \mathbb{N}$, but the theory applies for any totally ordered set. There are no real numbers in LF, so the tag set can be accurately modeled by $T = \mathbb{N} \times \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. The set is ordered lexicographically. Dispensing with real numbers means that some of the corner cases that arise in a generalized ultrametric semantics do not arise in LF. One subtlety that we do not escape, however, is the possibility of **Zeno systems**,

where one part of the system fails to advance time past a certain finite point while another part of the system proceeds beyond that point. Consider a program where one portion advances time only by microsteps and another by metric time. It can be shown that whether a given program is Zeno is undecidable; a clever demonstration of this has been given by Ben Lickly [126] who gives an example discrete-event program that is Zeno if the Collatz conjecture is false and non-Zeno if it is true. This example can also be easily be implemented in LF, as shown in Listing 3.22. Since Zeno systems are probably not useful, we will simply assume that our semantics does not include Zeno systems. The only thing remaining to do is prove that a program is modeled by a strictly contracting endofunction in the generalized ultrametric space. Determinism will then follow from the existence and uniqueness of a **fixed point** for this function.

First, we have to show that each reactor is indeed modeled by a function. This function has the form

$$F\colon (T \rightharpoonup V)^N \to (T \rightharpoonup V)^M,$$

where $(T \rightharpoonup V)$ is the set of all signals, N is the number of input ports, and M is the number of output ports. Some care is need here because a reaction contains arbitrary code in a target language, code that LF is not concerned with. If that code is nondeterministic, e.g., by invoking a random number generator seeded by the current time, then it is far from obvious how to model the reactor as such a function. But recognizing that our goal is to show the LF is deterministic (it does not introduce nondeterminism), not that the target language is deterministic, for each execution of the program, we can take the function to be the one determined by the particular *outcome* of every nondeterministic choice in the target language. This is analogous to the way the prior DE semantic models handle external inputs. For each execution, the function realized by each component is determined, in part, by the *particular* external inputs provided to that execution. For the example of the random number generator, we can consider the seed to be an external input. The function will be different for each execution of the program because the input will be different, but it will be a function nonetheless.

A similar strategy can be used to handle physical actions, which gets assigned a tag based on the current physical clock of the executing platform. The function realized by a reactor will depend on that tag, so that function will be different for every execution, but it is nevertheless a function, rendering the theory applicable. Hence, the tag, not just the value, of a physical action is considered an external input to the program. Given the inputs, including the tags assigned to physical actions, the behavior of the program will prove deterministic, an extremely valuable property (consider that it enables regression testing, for example).

A final subtlety is that LF allows reactions to overwrite an output produced by a previous reaction. Since these two output values have the same tag, this would seem to make it impossible to model an output signal as a function whose domain is the set of tags. However, because of the dependency analysis, which constrains the execution order of reactions, no other reactor will see the first value. Every other reactor sees only the final value at any tag, and hence there is no contradiction. That final value is the output from the function

Listing 3.22: Stuttering Zeno behavior exhibited if input disproves Collatz conjecture

```
target TypeScript {keepalive: true};
main reactor Collatz {
    logical action check:bigint
    physical action response:string
    preamble {=
        const readline = require('readline')
        const rl = readline.createInterface({
                input: process.stdin,
                output: process.stdout
        })
    =}
    reaction(startup) -> response {=
        rl.question("Enter an integer:\n", (answer:string) => {
                actions.response.schedule(0, answer)
            rl.close()
        })
    =}
    reaction(response) -> check {=
        actions.check.schedule(0, BigInt(response))
    =}
    reaction(check) -> check {=
        let n = check
        if (n !== undefined) {
            console.log(n)
            if (n <= 1n) {
                util.requestShutdown()
            } else {
                actions.check.schedule(0,
                    (n % 2n == 0n) ? n/2n : 3n*n +1n)
            }
        }
    =}
}
```

F (this also explains why we model reactors, not reactions as functions). We next need to show that for every execution, the F function for each reactor is contracting in a generalized ultrametric space. Following [42, 139, 156], we define the generalized ultrametric over the set $(T \rightharpoonup V)^N$ of N-tuples of signals to be a function

$$d : (T \rightharpoonup V)^N \times (T \rightharpoonup V)^N \to \Gamma$$

where Γ is the set of **down sets** of the tag set T, and N is a positive integer. For a particular pair of tuples of signals $\mathbf{s}_1$, $\mathbf{s}_2$, $d(\mathbf{s}_1, \mathbf{s}_2)$ is the largest down set of T where the restrictions of $\mathbf{s}_1$ and $\mathbf{s}_2$ to this down set are equal. In other words, $d(\mathbf{s}_1, \mathbf{s}_2)$ is the tag set of the **largest common prefix** of $\mathbf{s}_1$ and $\mathbf{s}_2$.

The set Γ is *totally ordered* by reverse set containment. Thus, for $\gamma_1, \gamma_2 \in \Gamma$, we write $\gamma_1 \leq \gamma_2$ if and only if $\gamma_1 \supseteq \gamma_2$. A function F modeling a reactor is a **contraction** if for all N-tuples $\mathbf{s}_1, \mathbf{s}_2 \in (T \rightharpoonup V)^N$,

$$d(F(\mathbf{s}_1), F(\mathbf{s}_2)) \leq d(\mathbf{s}_1, \mathbf{s}_2).$$

In words, the tag set of the common prefix of two possible outputs from the function is at least as big as the tag set of the common prefix of the two possible inputs that produce these outputs. This property is trivially satisfied by all LF reactors because outputs cannot depend on events with tags larger than that of the output. In other words, every reactor is causal (no output event depends on a future input event, one with a larger tag).

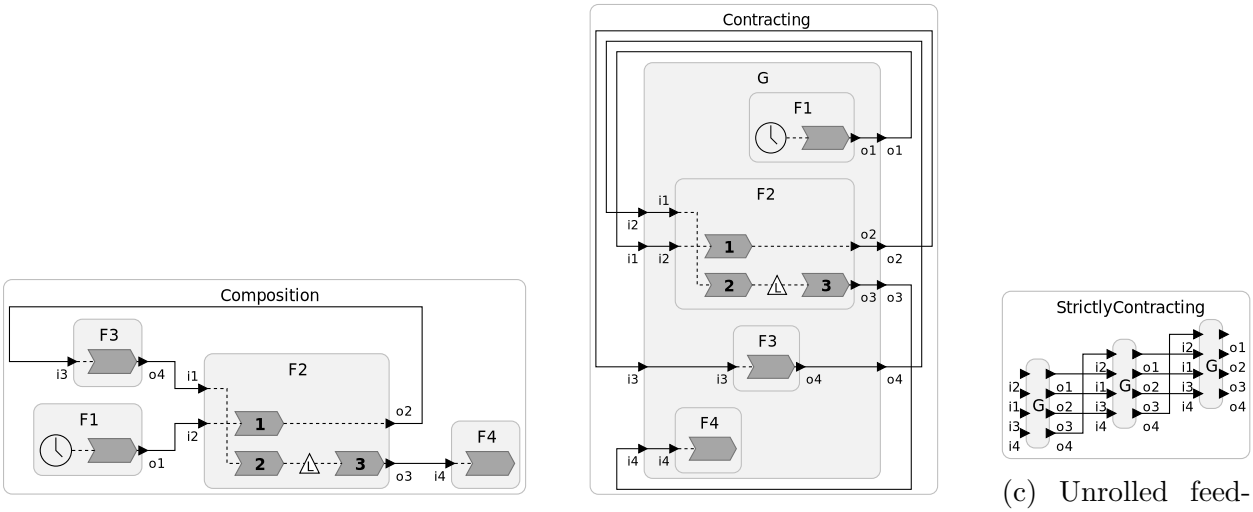
One final step is needed. Using the connections between ports to guide function composition, the individual functions F_r for each reactor r can be systematically composed to construct a function

$$G\colon (T \rightharpoonup V)^P \to (T \rightharpoonup V)^P,$$

where P is the total number of signals in the program and G describes the entire program. The procedure for constructing this function G is systematic (see [131], chapter 6).

An example fashioned after Figure 6.1 of [131] is given in Figure 3.4. Figure 3.4a shows a cyclic composition of four reactors producing four signals ($P = 4$). Each reactor is modeled by a function F_1 through F_4. These functions are assembled in parallel in Figure 3.4b to define an endofunction G that has the four signals as inputs and outputs. Four feedback connections then route each output to the corresponding input. The constraints of LF ensure that the graph of reactions (not reactors) is acyclic (any feedback loop in the dependency graph between reactions must include at least one microstep delay), and hence there always exists a finite unrolling (the function G applied to its own outputs some number N times) such that there is no path through the resulting graph of reactions from any input to the first G to any output of the last G. Since there is no such path, at each logical time, each output from G^N at each logical tag does not depend on any input at that logical tag. In the example, $N = 3$ is sufficient (see Figure 3.4c). In general, it is easy to show that N is no larger than the total number of reactions in the program.

Since any parallel composition of contracting functions is contracting, G is a contracting function. The function G^N, however, is **strictly contracting** because of the lack of direct



(a) Normal composition (b) Feedback composition (c) Unrolled feedback

Figure 3.4: Constructing a strictly contracting function G^N that models an LF program.

paths from any input to any output. Hence, every LF program can be modeled as a feedback loop with a strictly contracting function G^N mapping all signals to all signals. A classic fixed point theorem [175] tells us that such a function has exactly one fixed point, and hence there can be only one set of signals that satisfy the program. Hence, the program is deterministic. That fixed point theorem, however, is not constructive (it gives no way to find the fixed point). In [42], the classic Banach fixed point theorem, which applies to ordinary metric spaces, is generalized to apply to generalized ultrametric spaces. That theorem is constructive. The constructive procedure for finding the fixed point offers an **operational semantics** for LF. At the same time,the existence and uniqueness of the fixed point gives a **denotational semantics**. These two semantics match and hence are "fully abstract."

For completeness, one final observation is in order. LF permits the structure of programs to change at runtime through a mechanism called mutations (see Section 2.5). Logically, these mutations can be modeled as occurring between logical time steps because, at each time step, the mutations always precede any reaction that may be affected by the mutation. Semantically, this is termination of the execution of one deterministic program at the conclusion of a logical time step and starting a new deterministic program at the next logical time step. Two or more distinct functions G participate in determining the behavior of the program. As long as the time step is chosen deterministically and the mutation itself is a function of the inputs, the result is still a deterministic program.

We have to fight chaos, and the most effective way of doing that is to prevent its emergence.

Edsger W. Dijkstra

Chapter 4

Concurrency and Timing

Precise timing plays an important role in cyber-physical systems [124]. With their increasing computational demand, so is efficient exploitation of parallelism [12]. In order to effectively program these systems, there is a need for models with semantics that includes time, and we need runtime systems that are capable of harnessing the computing power of modern multi-core systems. The Reactor model and its Lingua Franca implementation is aimed at meeting these demands. Our approach is in contrast with today's general-purpose hardware and programming languages, where timing properties of software are *emergent* rather than specified, and exploiting concurrency is tedious due to the intrinsic difficulties dealing with threads [128] or endemic nondeterminism in coordination models like actors or service-oriented architectures [162]. The state-of-the-art in engineering realtime systems (which has not changed much since the early 2000s) relies heavily on overly detailed modeling and analysis or testing for the verification of timing properties [138, 47], but effectively testing software in the face of nondeterminism is challenging and sometimes infeasible.

Our goal is to chart a path toward a practice where timed behavior can be specified explicitly and its feasibility assessed statically, at compile-time. We are not there yet, but we see LF with its ability to specify timing behavior using first-class language constructs as an important step toward that goal. The key feature of the reactor model that enables this is its multiplicity of timelines and the relationship is establishes between them. Reactors leverage logical time, following classical synchronous-reactive principles, to cater deterministic responses to external stimuli that register as events with a tag derived from physical time (i.e., wall-clock time). This allows for the formulation of *deadlines*, which are bounds on the physical time permitted to elapse while reacting to events. While the LF toolchain currently performs no static WCET-analysis [180] or schedulability analysis [69, 18]—capabilities worth developing in the future—reactors are equipped with a fault handling mechanism for handling runtime violations of timing constraints. As faults can never be ruled out completely, we consider static analysis and fault handling complementary approaches to achieving robustness in time-critical systems.

The deterministic concurrency model of reactors is useful not only for systems that are time-critical. Any application that seeks to utilize multi-core architectures—even if it does

not care about timing at all—could benefit from the way reactors transparently exploit concurrency and help achieve reproducible program behavior.

4.1 Physical Actions in Reactive Systems

The use of physical actions and the distinction between physical and logical actions is sufficiently subtle that we feel compelled to offer an example illustrating the use of both. The example in Listing 4.1, which can be found in the Lingua Franca GitHub repository[1], implements a "reflex game" (a similar example was used by Berry and Gonthier [25]), where a user is presented with a prompt at a random time and asked to respond to the prompt by typing Return or Enter on the keyboard. The game then reports the number of milliseconds that elapsed between the prompt and the Return. If the user attempts to cheat by hitting return before seeing the prompt, the program detects it.

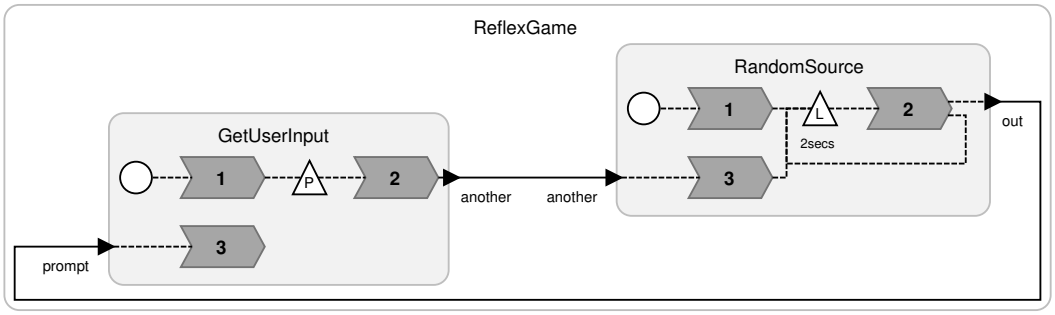


Figure 4.1: Diagram generated from the LF code in Listing 4.1.

The program consists of two reactors. `RandomSource` is responsible for generating a prompt at a random time. On Line 6, in response to **`startup`**, it uses the logical action named `prompt` to schedule a prompt to occur after two seconds plus an additional random delay specified by the function `rnd_time`. When that action occurs, it will print a prompt (Line 9). When an input event `another` occurs, it will schedule another instance of the `prompt` action (line 13). Using a logical action in this reactor makes sense because the reactor itself, not its physical environment, controls the timing of events.

The second reactor, `GetUserInput`, uses the pthreads library to start a thread that listens for keyboard inputs. The thread is started on Line 28 in response to **`startup`**. The new thread will, upon detecting that the user has typed Return, schedule the physical action `rspns` (Line 21). That action will be assigned a tag based on the current physical time as reported by the operating system or other time service on the execution platform. The reaction to `rspns` (Line 30) checks to see whether the user cheated and, if not, reports the response time. It then issues a request for another prompt (Line 38).

[1] https://repo.lf-lang.org/

Listing 4.1: Reflex game written in LF

```
target C;
reactor RandomSource {
    input another:bool; output out:bool;
    logical action prompt(2 secs);
    reaction(startup) -> prompt {=
        schedule(prompt, rnd_time(), NULL);
    =}
    reaction(prompt) -> out, prompt {=
        printf("Hit Return!");
        set(out, true);
    =}
    reaction(another) -> prompt {=
        schedule(prompt, rnd_time(), NULL);
    =}
}
reactor GetUserInput {
    preamble {=
        void* read(void* rspns) {
            while(1) {
                ... wait for Return key ...
                schedule(rspns, 0, NULL);
            }
        }
    =}
    input prompt:bool; output another:bool;
    physical action rspns; state prompt_time:time(0);
    reaction(startup) -> rspns {=
        pthread_create(..., &read, rspns);
    =}
    reaction(rspns) -> another {=
        if (self->prompt == 0LL) {
            printf("YOU CHEATED!\n");
        } else {
            int t = (get_logical_time() - self->prompt) / MSEC(1);
            printf("Time in ms: %d\n", t);
            self->prompt_time = 0LL;
        }
        set(another, true);
    =}
    reaction(prompt) {=
        self->prompt = get_physical_time();
    =}
}
main reactor ReflexGame {
    p = new RandomSource(); g = new GetUserInput();
    p.out -> g.prompt; g.another -> p.another;
}
```

Using a physical action for the second reactor makes sense because the timing of the events of this action are determined by the physical environment, not by the reactor itself. LF ensures that the tags assigned to these events will not appear "in the past." In other words, it ensures that all reactors see events in timestamp order. The precision with which these logical timestamps match physical time, of course, will depend on the properties of the real-time clock on the execution platform.

Deadlines

The `GetUserInput` reactor in Listing 4.1, which turns keystrokes into tagged events through a physical action, is an example of a reactor that wraps a sensor. A typical use case for such a component would be to integrate it into a control system such that it triggers some computation, the result of which ultimately drives an actuator. In our reflex game, the "actuator" just prints to `stdout`. (Line 35). Such control systems are prevalent in automotive applications, fly-by-wire systems in aircraft, and really any kind of cyber-physical system. What these applications typically have in common is that they are subject to a specification that imposes bounds on the maximum latency between sensing and actuation. In an automotive brake system, for instance, the physical time that elapses between the moment of pressing the brake pedal and the brakes being applied has to be bounded in order to guarantee a braking distance that is considered safe (see Section 2.4).

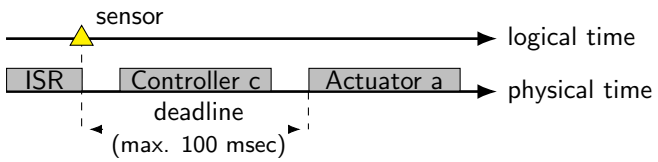


Figure 4.2: A deadline defines the maximum delay between the logical time of an event and the physical time of the start of a reaction that it triggers.

We call these bounds deadlines. A deadline Δ specifies a time interval such that the reaction to the input `in` with tag $g = (t, m)$ is required to be invoked before physical time, as measured on the local platform, exceeds $t+\Delta$. In other words, before invoking the reaction to input `in` at a logical time g, the LF runtime system checks the local physical time T; if $T \leq t + \Delta$, then it invokes the reaction as usual (Line 13). Otherwise, it invokes the code at Line 15 that handles a deadline miss. That code could, for example, raise an alarm and/or change the system to operate in some sort of safe degraded mode.

The program in Listing 4.2 illustrates how the end-to-end latency between a sensor and an actuator can be bounded by a deadline. The program instantiates two reactors `c` and `a`, instances of `Controller` and `Actuator` respectively. The physical action `sensor` on Line 2 will be triggered by an asynchronous call to the SCHEDULE procedure, for example, within an interrupt service routine (ISR) handling the sensor (that code is not shown). The action will be assigned a tag based on what the physical clock indicates when the ISR is invoked.

Listing 4.2: Bounded end-to-end delay between a sensor and an actuator

```
1  reactor Controller {
2      physical action sensor:int;
3      output y:int;
4      // ...
5      reaction(sensor) -> y {=
6          int control = calculate(sensor_value);
7          set(y, control);
8      =}
9  }
10 reactor Actuator {
11     input x:int;
12     reaction(x) {=
13         // Time-sensitive code
14     =} deadline(100 msec) {=
15         printf("*** Deadline miss detected.\n");
16     =}
17 }
18 main reactor Composite {
19     c = new Controller();
20     a = new Actuator();
21     c.y -> a.x;
22 }
```

That tag, therefore, is a measure of the physical time at which the sensor triggered. The reaction to `sensor`, on Line 6, performs some calculation and sends a control messages to its output port. Line 21 connects that output to the input `x` of the actuator.

The actuator's reaction to the input `x` declares a deadline of 100 ms on Line 14 followed by a deadline violation handler. If this reaction is not invoked within 100 ms of the tag of the input, as measured by the local physical clock, then rather than executing the time-sensitive code in the reaction, the deadline violation is handled. The deadline, therefore, is expressing a requirement that the calculation on Line 6 (plus any overhead) not take more than 100 ms (in physical time). This relation across timelines is illustrated in Fig. 4.2.

A deadline in an LF program has two roles. First, it provides a hint to the scheduler. Our multi-core scheduler for the C target implements an earliest-deadline-first (EDF) scheduling strategy [39], where every reaction upstream of a reaction with a deadline inherits its deadline (or an earlier deadline if there are more than one downstream reactions with deadlines). Second, it provides a mechanism for providing a fault handler, a body of target code to invoke if the deadline is violated. Note that the deadline construct in LF admits nondeterminism. The program will be deterministic only if the deadlines are not violated. Whether the deadline is violated or not depends on factors outside the semantics of LF.

If the tag g for the event presented to input `in` was ultimately derived from a physical action, then the deadline in Listing 4.2 specifies an end-to-end deadline between sensing

and actuation. The deadline may be violated, for example, by excessive execution times of reactions in the path to `in`, or by poor scheduling decisions that failed to take into account the deadline. To provide assurance that deadlines are not violated requires estimates of worst-case execution time (WCET) of code fragments. LF's architecture naturally breaks down code into fragments, the reactions, that may prove more amenable to WCET analysis than arbitrary programs. An excellent survey of the state of the art in WCET analysis is provided by Wilhelm et al. [216]. We have not implemented such analysis, but see adding WCET estimation tools to our compiler toolchain as potential future work.

Logical Time Delays

A **logical time delay** between two reactions can be implemented using a logical action. As a convenience, LF allows for connections to be annotated with an `after`-clause that specifies a time delay. Such delay effectively shifts a produced output along the logical time line. As such, this mechanism can be used to reduce the amount by which logical time lags physical time, and account for the execution time of reactions. By choosing the logical delay between two reactions connected to one another via ports—a producer and a consumer—such that the logical delay exceeds the worst-case execution time (WCET) of the producer, the tags of the events are always greater than the physical time at which they are produced. This effectively assigns a logical execution time (LET) [93] to the producer, allowing the execution of the consumer to be timed more precisely with respect to physical time.

4.2 Runtime Scheduling and Real-Time Constraints

The execution algorithm for reactors explained in Section 2.7 honors the data dependencies that exist between reactions, but it leaves room for scheduling decisions that may affect the system's performance, both in terms of latency, throughput, and its ability to meet deadlines. Specifically, on Line 16 of Algorithm 12 (the DOSTEP procedure), the SELECT procedure that picks the next reaction to be executed from the pool of "ready" reactions remains unspecified.

More generally, several key implementation details of the runtime algorithms discussed in Section 2.7 are intentionally abstracted with mathematical notation. To get a better understanding of some trade-offs an actual runtime scheduler is able to make, a bit more detail is needed. Let us first assume that event queue $\mathcal{Q}_E$ and reaction queue $\mathcal{Q}_R$ are implemented as a **priority queue** Using priority queues allows us to efficiently ordering events and reactions, without having to perform expensive graph searches at runtime, for instance. While $\mathcal{Q}_E$ simply has to order events by their tag, it is less clear how the priority of reactions in $\mathcal{Q}_R$ should be encoded. There are several options. The most straightforward way to prioritize reactions is to assign them a numerical value based on their position in what is called a **topological sort**. A topological sort of the reaction graph $\gamma_N(r)$ of some top-level reactor r consists of a linear ordering of the graph's vertices such that for every

directed edge (n', n) from reaction n' to reaction n, n comes before n' in the ordering. There is often more than one ordering possible that satisfies this topological sorting constraint.

Algorithm 17 Assign levels to all reactions in a top-level reactor r

```
1: procedure ASSIGNLEVELS(r)
2:     (V, E) ← γ_N(r)      ▷ Reaction graph; vertices are reactions and edges denote dependencies
3:     for each n ∈ V do
4:         l(n) ← 0                                  ▷ Initialize the level of all nodes to zero
5:     end for
6:     S ← LIST({n ∈ V | ∄n' ∈ V . (n, n') ∈ E})                  ▷ Create list of start nodes
7:     while |S| > 0 do
8:         n ← POP(S)                          ▷ Remove one element from the list of start nodes
9:         for each n' ∈ V . (n', n) ∈ E do          ▷ Iterate over reactions that depend on n
10:            l(n') ← max(l(n'), l(n) + 1)      ▷ Assign level one higher than upstream neighbor
11:            E ← E \ {(n', n)}                    ▷ Remove visited edge from the graph
12:            if ∄n' ∈ V . (n, n') ∈ E then  ▷ Check whether unvisited upstream neighbors exist
13:                PUSH(S, n)                        ▷ If not, add the node to the start list
14:            end if
15:        end for
16:    end while
17:    if |E| > 0 then                ▷ If edges remain in the graph, there must be a cycle
18:        error: Cycle in graph.
19:    end if
20: end procedure
```

Since we are interested in executing reactions in parallel whenever the absence of data dependencies allows us to do so, we are more interested in establishing a partial order than the total order that we would obtain from an ordinary topological sort. With a slight adjustment of any ordinary topological sort algorithm, we can assign **levels** instead of positions in a list. This increases parallelism because any two reactions of equal depth can now execute in parallel. In the LF compiler, we use an adapted variant of Kahn's algorithm, shown in Algorithm 17, to assign levels to reactions. This is still a conservative approximation of the dependencies in our reaction graph, however. If there exists a dependency between n' and n, then the level of n must be less than the level of n', denoted as $l(n) < l(n')$, but if $l(n) < l(n')$, then this does not imply that there must exist a dependency (n', n) in the reaction graph. More parallelism can be exposed through a more advanced encoding, which we discuss in Section 4.3.

Levels alone, however, are already sufficient to exploit parallelism in common patterns such as fork-join parallelism and pipelines. Figure 4.3 shows a typical scatter/gather pattern, where all the Computation reactors can be executed in parallel provided there is a sufficient number of worker threads to do so. In Fig. 4.4, a chain of reactions is triggered by a timer with a specified period. Each reaction produces an output event that is logically simultaneous with its input, but each stage of the pipeline is broken up by a delay specified

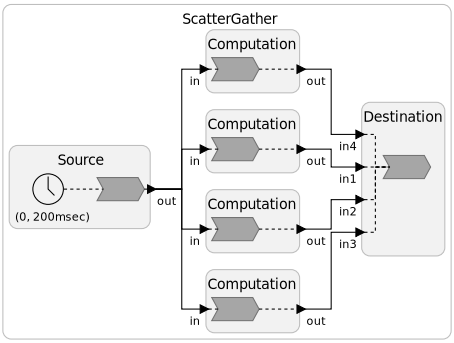


Figure 4.3: A diagram of an LF program realizing a typical scatter/gather pattern.

by the parameter period. This effectively breaks up the reaction graph into disconnected subgraphs. At each tag, there is no dependency between any two reactions in the pipeline, so they can all be executed in parallel.

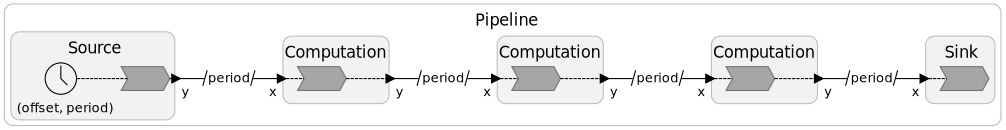


Figure 4.4: A diagram of a pipeline pattern in LF; each stage executes in parallel.

Earliest-deadline-first Scheduling

While ordering reactions by level accounts for their dependencies, it does not account for deadlines. If there is no risk of violating any dependencies, then it would pay off it execute a reaction with an earlier deadline, because this would make it more likely for the deadline to be met. This strategy was formalized by Liu and Layland [136] in 1973 and is since known as Earliest Deadline First (EDF) scheduling, probably the most common dynamic priority scheduling algorithm for real-time systems.

We have implemented a non-preemptive version of EDF scheduling in our C-based runtime by changing the sorting criterion of $\mathcal{Q}_R$ to take into account deadlines. To do this, a preprocessing step is carried out in the compiler to ensure that each reaction inherits the earliest deadline among all of its downstream reactions in the reaction graph. The propagation algorithm used for this is provided in Algorithm 18. Reactions without a specified deadline and no downstream deadlines will have a deadline of ∞ (which in any practical realization is the maximum value that can be expressed with the used data type).

For efficiency, we pack the deadline and the level in a single unsigned 64-bit integer. The most significant 48 bits are reserved for the deadline and the remaining 16 bits are used for the level. All time values in the C runtime have a nanosecond precision, meaning that with 48 bits a deadline can range from 10^{-9}s to roughly 2.8×10^5s (almost half a week). With 16 bits to encode the level, we allow a maximum of 65536 levels.

Algorithm 18 Propagate deadlines between reactions in top-level reactor r

```
1: global variables
2:     (V, E) ← γ_N(r)      ▷ Reaction graph; vertices are reactions and edges denote dependencies
3: end global variables
4: procedure ASSIGNDEADLINES(r)
5:     for each n ∈ V . Δ(n) = ⊥ do
6:         Δ(n) ← ∞                    ▷ Assign default deadline to reactions without a deadline
7:     end for
8:     for each n ∈ V . Δ(n) ≠ ∞ do
9:         PROPAGATEDEADLINE(n)                ▷ Propagate specified deadlines upstream
10:    end for
11: end procedure
12: procedure PROPAGATEDEADLINE(n)
13:    for each n′ ∈ V . (n, n′) do
14:        if Δ(n) = ⊥ then
15:            Δ(n′) ← min(Δ(n′), Δ(n))      ▷ Inherit deadline if smaller than the current
16:        end if
17:        PROPAGATEDEADLINE(n′)                ▷ Continue to propagate upstream
18:    end for
19: end procedure
```

Preemption

The scheduler in the C runtime is currently non-preemptive, meaning that once started, each reaction runs to completion without any interruptions from other reactions. This can negatively impact the feasibility of schedules (i.e., the ability to meet deadlines). Specifically, without preemption, there is no possibility to suspend the execution of reactions with a later deadline in favor of ones with an earlier deadline that may be released while all worker threads are occupied by less urgent reactions. When combined with a preemptive thread scheduler (and a number of worker threads that exceeds the number of cores), the runtime could dynamically change thread priorities to achieve preemption.

4.3 Exposing More Parallelism

It would be prohibitively expensive to walk the reaction graph at runtime to discover dependencies between any two reactions that are ready to execute, and ordering reactions by their level (i.e., a reaction with no dependencies has level 0, it's immediate downstream neighbors have level 1, etc.) does not expose all parallelism. For instance, could add a parallel path from the Source to Destination reactor that has not one but two reactions (n_1 and n_2) in sequence, which jointly would take about as much compute time as the single reaction in each Computation reactor. In that case, level $l(n_1) = 1$ and thus would be allowed to execute in parallel with the other reactions from Computation, but $l(n_2) = 2$, meaning the second

reaction would be forced to wait for all parallel reactions to conclude, even though there clearly is no dependency that would require this.

We can improve on this with a scheme similar in spirit as the fast dynamic casting algorithm by Gibbs and Stroustrup [83]. Where the correctness of dynamic casting depends on the existence a certain inheritance relationship, the correctness of selecting a next reaction to execute hinges on the absence of certain data dependencies. Rather than walking the inheritance tree, Gibbs and Stroustrup assign cleverly chosen IDs (prime numbers) to each class, and use the modulo operator at runtime to figure out whether a cast is legal or not, which is obviously much cheaper. Similarly, we assign IDs to reactions at compile time in our scheme. But instead of primes, we use carefully chosen binary numbers, and instead of the modulo operator, we use a bitwise AND to determine whether two reactions have a directed path between each other; if they do, we say they are part of the same **chain**. We denote the **chain ID** of a reaction n as $\mathfrak{i}(n)$. A reaction n' only truly depends on another reaction n if the following predicate is true:

$$l(n) < l(n') \wedge (\mathfrak{i}(n) \mathbin{\&} \mathfrak{i}(n')) \neq 0,$$

where & denotes bitwise AND. The cost of evaluating this predicate at runtime is extremely low, and it can be evaluated lazily, meaning that the right-hand side only has to be evaluated when the left-hand size evaluates to true.

Whereas the algorithm for assigning levels works from the roots of the dependency tree toward its leaves, the traversal that assigns chain IDs (see Algorithm 19) works in the opposite direction. The goal is to compute a **path cover** that consists of all paths between any pair of vertices in the reaction graph consisting of a leaf node (a reaction that no reaction depends on) and a root node (a reaction with no dependencies), and to assign a unique ID to each such path. First, we assume some value for w, the width of the bitstring that encodes the chain ID. We use 64 bits in our C runtime. Furthermore, we maintain a counter, c, that we increment with each new leaf node that we visit. For each leaf, we create a fresh chain ID that is simply $2^{c \bmod w}$, or, using binary operators: `1 << c % w`. We then recursively propagate that ID in a depth-first fashion. During that process, a mask gets constructed for every visited node based on a bitwise OR of the masks constructed by subsequently visited nodes. For any dependency that a node has beyond one, a new chain ID is allocated (i.e., c is incremented), and then that new chain ID gets propagated. No new chain IDs are allocated during the traversal unless branching occurs. When all the masks have returned after visiting a node's dependencies, its current ID is combined with the constructed mask, again using a bitwise OR. This step ensures that each reaction has at least one bit in common with the ID of reactions that it depends on.

Let us consider the dependency graph depicted in Figure 4.5. Assuming we have already assigned a level to each node in the graph using Algorithm 17, we now need to assign IDs. We start out with $c = 1$, and first visit H, because $l(H)$ is greater than the level of any other leaf node in the graph. We then proceed to visit F, E, C, and B. Once we reach B, we stop the traversal and start backtracking. The reason for this is that there are other paths from

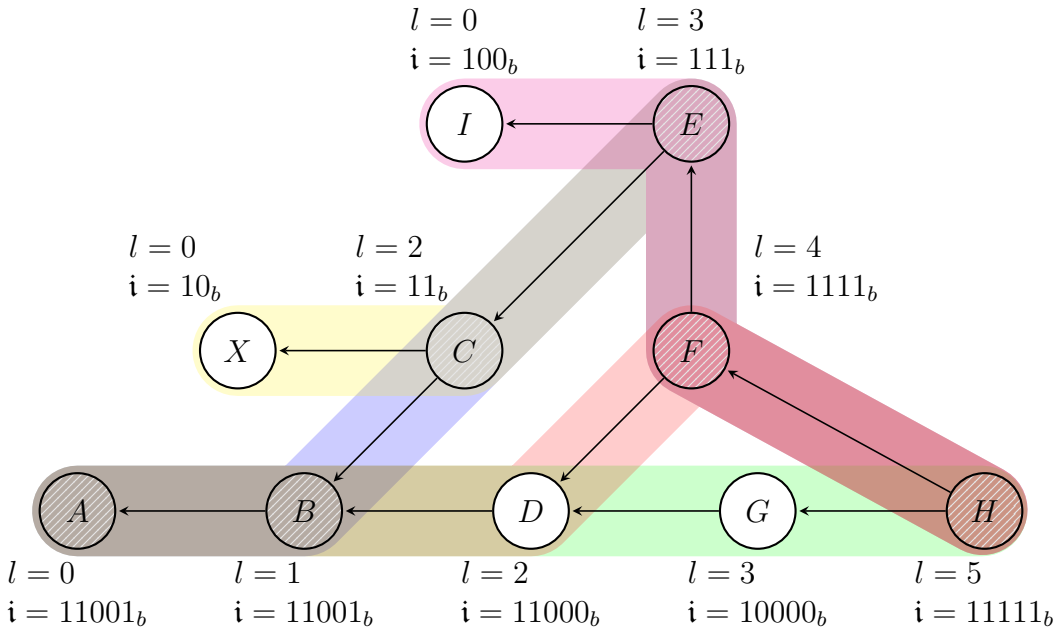


Figure 4.5: An example reaction graph with assigned levels and IDs.

H that lead to B—these paths have to be explored first. Before doing that, we temporarily assign $\mathfrak{i}(B) \leftarrow 1_b$.

Arriving back in C, we explore a new branch, meaning we increment c, visit X, and assign $\mathfrak{i}(X) \leftarrow 10_b$. We return 10_b back to C and assign $\mathfrak{i}(C) \leftarrow 11_b$. We then further backtrack to E, visit I to which we assign $\mathfrak{i}(i) \leftarrow 100_b$ after having incremented c. Then 100_b is returned, so we assign $\mathfrak{i}(E) \leftarrow 111_b$. Arriving back at F, we have another branch to explore, hence we increment c and visit D. But D has a remaining visit count of 1, so we temporarily set $\mathfrak{i}(D) \leftarrow 1000_b$. We return, backtrack to F, and assign $\mathfrak{i}(F) \leftarrow 1111_b$.

Finally, we backtrack to H, after which we explore the last path. We increment c once more, and we visit G, and then D. All paths to D have now been covered, so we continue to visit B, which now also has a visit count of zero, meaning we proceed to A and assign $\mathfrak{i}(A) \leftarrow 11001_b$, which is a combination of 1_b that was stored in F, 1000_b that was stored in G, and 10000_b the identifier associated with the current path. As we backtrack, the last assignments to be made are $\mathfrak{i}(B) \leftarrow 11001_b$, $\mathfrak{i}(D) \leftarrow 11000_b$, $\mathfrak{i}(G) \leftarrow 10000_b$, and $\mathfrak{i}(H) \leftarrow 11111_b$. The longest chain in graph in Figure 4.5 consists of the shaded nodes H, F, E, C, B, and A. We have four more chains: (H, G, D, B, A), (H, F, D, B, A) (H, F, E, C, X), and (H, F, E, I).

For a reaction graph that has more than w chains in it, the modulo operator used on Lines 15 and 37 facilitates the reuse of chain IDs, which could limit the amount of exposed parallelism. In a less conservative approach, one could simply use a larger w, which would come at the cost of having to do multiple bitwise ANDs at runtime if w exceeds the word size of the architecture. Heuristics could also be used to find with a more economical assign-

ment than the one achieved with Algorithm 19. Specifically, our algorithm would naively assign different IDs to each chain in the reaction graph of the example in Figure 4.3, which is unnecessary because the reactions of each Computation reaction can already execute in parallel by virtue of them all having the same level. We leave the implementation of such improvements for further work.

Note that we avoid visiting chains more than once by breaking off the recursion when there are still other paths left that need to be visited in order to determine all bits in the chain ID that is to be propagated to upstream reactions. Each leaf node is visited once, and every other node will be visited once for each path from a leaf node that reaches it. Only after all dependent reactions have been visited will the propagation of chain IDs to upstream reactions continue. Since the number of nodes to visit is bounded by the number dependencies that each node has (and in the worst case each node depends on every other node), the worst-case complexity of Algorithm 19 is $\mathcal{O}(|V|^2)$.

4.4 Further Optimizations

There are more opportunities for runtime optimizations. We discuss some of them.

Immediate Reactions

When a reaction sets a value on a port, any reactions that are triggered as a consequence are added to the reaction queue $\mathcal{Q}_R$. Once the reaction concludes, $\mathcal{Q}_R$ is checked to find out which reaction to execute next. Sometimes the next reaction is precisely the reaction that was pushed onto $\mathcal{Q}_R$ a moment earlier by the preceding reaction. If the preceding reaction triggered exactly one reaction, then the overhead associated with interacting with $\mathcal{Q}_R$ can be avoided by executing the triggered immediately, bypassing $\mathcal{Q}_R$ altogether. In the multi-threaded runtime, this avoids acquiring a mutex lock—if there are no deadlines in the program. If there are deadlines, then the queue still must be checked to determine the earliest deadline in $\mathcal{Q}_R$ (or else this optimization could violate the EDF scheduling policy). This optimization has been implemented in the C runtime.

Reacting Ahead of Physical Time

Generally, an event should not trigger any reaction before physical time has surpassed the time value of its tag. This prevents a scenario where an event gets scheduled with a tag smaller than the tag of an event that has already been released into the runtime system. This, however, is a conservative rule that can be relaxed under certain circumstances. For instance, if there are no physical actions in the program, then this scenario will simply never occur. But even if there are physical actions, their minimum spacing might present opportunities for safely moving ahead of physical time for limited time intervals. For example, if a physical action a has a minimum spacing $s(a)$ of 20 ms, and the last event $\mathcal{L}(a) = g$, then we know

Algorithm 19 Assign chain identifiers to reactions in a top-level reactor r

```
1: global variables
2:     (V, E) ← γ_N(r)      ▷ Reaction graph; vertices are reactions and edges denote dependencies
3:     c ← 0                ▷ Global branch count; increases during graph traversal
4:     const w ← 64         ▷ Word size; 64 bits by default
5: end global variables
6:
7: procedure ASSIGNCHAINIDS(r)
8:     for each n ∈ V do
9:         VISITCOUNT(n) ← |{n' ∈ V | (n', n) ∈ E}|      ▷ Initialize visit count
10:    end for
11:    leafs ← {n ∈ V | ∄(n', n) ∈ E}
12:    while leafs ≠ ∅ do
13:        highest ← {n ∈ leafs | ∀n' ∈ leafs . l(n) ≥ l(n')}      ▷ Next nodes to visit
14:        for each n ∈ highest do      ▷ Visit leaf nodes, highest level first
15:            PROPAGATEID(n, 2^(c mod w))      ▷ Propagate ID based on branch count
16:        end for
17:        leafs ← leafs \ highest
18:    end while
19: end procedure
20: procedure PROPAGATEID(current, chainID)
21:    c ← c + 1      ▷ Increment branch count
22:    mask ← chainID      ▷ Bitmask to be adjusted based on upstream chainIDs
23:    upstream ← {n ∈ V | (n, current) ∈ E}      ▷ Find upstream neighboring nodes
24:    first ← true
25:    id ← i(current) | chainID      ▷ Bitmask to be passed as chainID to upstream nodes
26:    VISITCOUNT(n) ← VISITCOUNT(n) − 1
27:    if VISITCOUNT(n) > 0 then
28:        i(current) ← id      ▷ Update node and return
29:        return chainID
30:    end if
31:    while upstream ≠ ∅ do
32:        nearest ← {n ∈ upstream | ∀n' ∈ upstream . l(n) ≥ l(n')}      ▷ Next nodes to visit
33:        for each node ∈ nearest do      ▷ Visit upstream neighbors, highest level first
34:            if first then
35:                first ← false
36:            else
37:                id ← 2^(c mod w)      ▷ Recalculate chainID passed to upstream neighbors
38:            end if
39:            mask ← mask | PROPAGATEID(node, id)      ▷ Update mask (bitwise OR)
40:        end for
41:        upstream ← upstream \ nearest
42:    end while
43:    i(current) ← i(current) | mask      ▷ Update chainID of current node (bitwise OR)
44:    return mask
45: end procedure
```

that no event will appear on a with a tag earlier than $g' = g + (20\,\text{ms}, 0)$, meaning it would be safe to handle an event with a tag g'' provided that $g'' < g$. When this optimization is applied, triggered reactions of realtime reactors (see Section 3.5) would simply have to wait in the reaction queue until $T > \pi_1(g'')$, also blocking progress of any triggered reactions that depend on them. This would still allow for an amount of "precomputation" upstream of realtime reactors, thereby tightening the realtime reactors' synchronization to physical time.

Relaxing the Barrier Synchronization

Another aspect of the default execution algorithm for reactors that limits the amount of work that can be done in parallel is the **barrier synchronization** that occurs after each synchronous-reactive step. While effective, this is relatively crude measure to ensure that each reactor observes events in tag order and has its reactions triggered accordingly. For instance, a reaction with no dependencies (e.g., driven by a timer) could, in principle, ignore the barrier and precompute future output values as long as it would not present those to downstream receivers ahead of time. And by "ahead of time" we mean "before its reactions to events with earlier tags have concluded." What events with earlier tags may appear (and whether these, too, can be precomputed) depends on information that can be gleaned from the structure of the program. Again, the minimum spacing property of actions can open up a time window during which it is known that particular events will be absent. A reaction could safely bypass the barrier if all its dependencies are either precomputed or known to be absent.

It remains an open question how opportunities for precomputation can be exploited efficiently. To an extent, the decentralized coordination scheme for federated reactors described in Section 5.2 already realizes some of these optimizations. Under decentralized coordination, federates advance time independently (albeit subject to constraints), but all reactors within each federate are still synchronized using a barrier. Of course, federated execution also comes at the cost of serialized communication through sockets instead of communication through shared memory. Thus, if this kind of approach were to be leveraged for performance gain, then the amount of extra parallel computation would have to outweigh the communication overhead of federated execution.

Static Scheduling

For programs that are limited to a restricted subset of behaviors, such as synchronous dataflow [120], an optimized static schedule could substitute the dynamic runtime scheduler. This could be done automatically in the LF compiler directed by some target property. We leave this for future work.

4.5 Subroutines

Subroutines among some of the most rudimentary and powerful programming constructs that exist. Virtually every programming language features them in some shape or form. Sometimes they are called routines, subprograms, functions, methods, or procedures, but they are all meant to do the same thing: decompose a complex programming task into smaller, simpler steps. When used properly, they also reduce code duplication and improve code readability. Implementing an interaction between reactors that resembles the invocation of a subroutine is possible, but it comes with a certain amount of awkwardness. Let us consider the example in Figure 4.6 that is inspired by a situation that is commonly found in control logic; the need to check for some safety condition before carrying out some requested operation. This pattern could be applied, for example, in a stall a prevention mechanism of a fixed-wing aircraft, where Caller instance foo receives input from the control wheel and Callee instance bar reports the angle of attack. In an airlock aboard a spacecraft, foo could be responding to a button press requesting the door to open, and bar would report the pressure inside the pressure vessel. The basic idea is that foo needs to momentarily gain access to the state of bar before it can continue. This is the reactor equivalent of a subroutine.

As shown in Figure 4.6, the response to the physical action in foo has to be split into *two* reactions, and whatever state computed in the first reaction that is necessary in the second reaction needs to be stored in a state variable (this is a problem also referred to as stack ripping). The response from the bar is fed back to foo via an input port, establishing a feedback loop between the two reactors. In a situation like this, it would be more attractive if foo could get a response from bar without this level of indirection, like one would achieve with an ordinary subroutine.

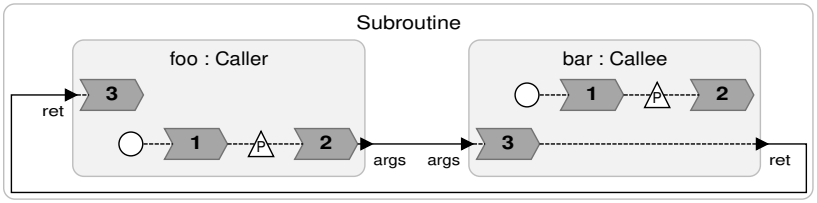


Figure 4.6: The reactor equivalent of a subroutine.

It is possible to extend reactors with such a mechanism. We have implemented this extension in our TypeScript runtime, and it works as follows. In addition to ordinary input and output ports, we distinguish a **caller port** and a **callee port**. Unlike inputs and outputs, these new ports are bidirectional and thus have a type associated with each direction. In our implementation, the classes `CallerPort<A,R>` and `CalleePort<A,R>` each have two type variables of which `A` stands for "arguments" and `R` stands for "return value." A caller can be connected to a callee if and only if $A_{caller} \preceq A_{callee}$ and $R_{callee} \preceq R_{caller}$, following the usual contravariant subtyping rule for functions. Instead of invoking SET, a reaction calls INVOKE on a caller port to directly execute the reaction triggered by the callee port

that the caller port happens to be connected to. This pattern is much closer to an ordinary subroutine. It does not involve cycles in the connection topology and there is no necessity for stack ripping. We do preserve our composition mechanism based on ports, so the complete separation of implementation and composition is kept; reactors remain fully agnostic the counterparts they may be composed with, even though they may invoke their reactions directly. Figure 4.7 shows a version of our example that uses caller and callee ports.

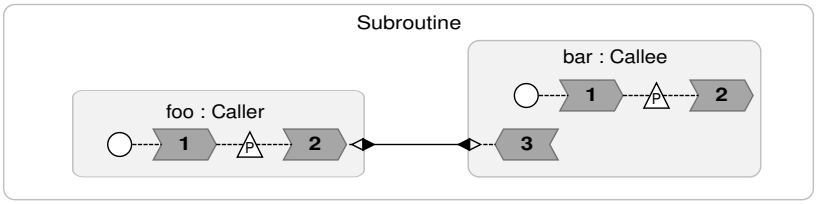


Figure 4.7: An alternative implementation of Figure 4.6 using a caller and callee port.

Of course, connections between **callers** and **callees** imply dependencies, which are necessary to preserve determinism. These dependencies are different from the dependencies implied by connections between regular input/output ports. While ports are used as an intermediary, let us use the term "caller reaction" for the reaction that calls INVOKE and "callee reaction" for the reaction that is executed in turn to produce the return value. A callee reaction is triggered by a single callee port has no effects (i.e., it produces no outputs). The callee reaction provides its return value to the caller by calling ANSWER on its trigger. Control returns to the caller when the callee reaction is done executing.

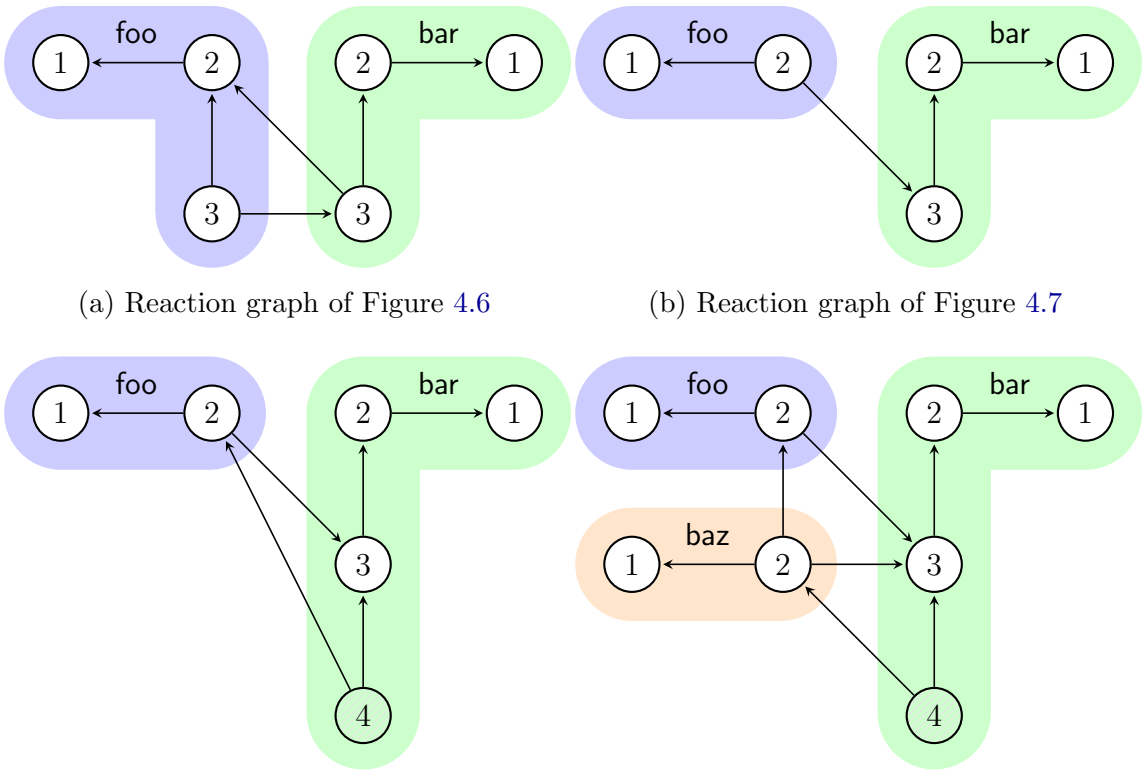
Connections between caller ports and callee ports imply the following dependencies:

1. Any caller reaction must depend on their corresponding callee reaction. This ensures that any reactions that have precedence over the callee reaction due to reaction priority will execute first. This is necessary because the state of the reactor that contains the callee reaction must have settled before the callee reaction is invoked, or else a race condition would arise.

2. All concurrent caller reactions *that invoke the same callee* must have dependencies between them. This ensures that they enjoy mutual exclusivity and execute in a deterministic order.

3. If there exists a reaction over which the callee has precedence due to reaction priority, then that reaction has to have a dependency on *the last caller reaction that invokes the preceding callee*, again to avoid a race condition.

Figures 4.8a and 4.8b show the reaction graphs of Figures 4.6 and 4.7, respectively. The direct invocation of bar.3 by foo.2 leads to a dependency inversion; bar.3 in Figure 4.8a depends on foo.2 whereas in Figure 4.8b foo.2 depends on bar.3. This particular dependency prevents foo.2 from executing before the state of bar has settled. If we add an extra reaction,



(a) Reaction graph of Figure 4.6

(b) Reaction graph of Figure 4.7

(c) Reaction graph in Figure 4.8b with one extra reaction in the Callee

(d) Reaction graph in Figure 4.8c with one extra Caller

Figure 4.8: Reaction graphs explaining the dependencies in subroutine-like interactions.

bar.4 that depends on bar.3, then that reaction will also have to depend on foo.2, as shown in Figure 4.8c. Finally, in Figure 4.8d, we see that if we add more callers that invoke bar.3, then their calling reactions will have to be arranged in dependency chain, with bar.4 depending on the last node in that chain. Support for caller and callee ports will require adaptations to CONNECT and DISCONNECT to account for these type of changes to the reaction graph.

Deadlock Freedom

Causality loops will prohibit certain configurations, but a configuration without causality loops is also deadlock-free, generally a non-trivial property in concurrent systems with blocking procedures [10, 43, 169].

Performance

The avoidance of stack ripping and the inlining of the callee reaction (it bypasses the reaction queue), can also lead to formidable performance improvements. We found that using caller

and callee ports in the PingPong benchmark of the Savina actor benchmark suite [99] lead to a 6× speedup, measured in our TypeScript runtime.

4.6 Performance Benchmarks

It is too early for a full-fledged performance analysis of LF. We have a reasonably well-developed suite of regression tests, replicated for each target language to the extent that the tested features are implemented in the respective targets, but the tests are concerned with correctness, not performance. So far, our most mature targets are C and C++. Part of our motivation to focus on these relatively low-level languages is to achieve a runtime implementation with minimum overhead. We will discuss a preliminary performance evaluation that is performed on a Dell® PowerEdge R730 equipped with 6-core Intel® Xeon® CPU E5-2643 v3 @ 3.40GHz and 96GB of memory. The operating system is Arch Linux.

Our most rudimentary performance indicator is a regression test that also provides a measure of runtime overhead. This test has one reactor with single output connected to another reactor with a single input. Each of these reactors has a state variable `count`; the upstream reactor has it initialized to 0 and the downstream reactor has it initialized to 1. The upstream reactor has a reaction triggered by a timer, which increments the reactor's `count` and assigns its value to its output port. The downstream reactor has a reaction, triggered by its input port, that compares the input against its own `count` and increments that `count` if it matches. The program exits when the downstream reactor's `count` has reached 1×10^8, i.e., after 2×10^8 reactions have executed. When executed on our evaluation system, this program executes in 793 ms (averaged over ten runs), which translates into 40 ns per reaction invocation.

Because reactors are a new programming paradigm, it is not immediately obvious what would be an appropriate baseline to compare against. Since actors are strongly related and known for their performance and widespread use, it would be interesting to compare against those. The asynchronous message passing of actors is very different from the synchronous communication between reactors, and it would be tempting to assume that the synchronization of reactors would impose a considerable performance cost compared to the much less constrained communication patterns between actors. Perhaps somewhat surprisingly, we found that this is not necessarily the case. We have started to evaluate the performance of our C runtime by implementing a subset of the Savina actor benchmark suite, which is a widely cited set of benchmarks developed by Imam and Sarkar [99]. The Savina suite features three categories: micro benchmarks, concurrency benchmarks, and parallelism benchmarks. In this preliminary evaluation we discuss one benchmark from each of these categories. A more comprehensive evaluation of the performance of the C++ runtime has been conducted by Hannes Klein in his Bachelor thesis [109], which covers a much larger subset of the Savina suite than the small sampling we discuss here. Thus far, no serious effort has been made to optimize our reactor runtime implementations, so we expect the benchmarking results reported in [109] and this thesis to leave significant room for performance improvements.

One of the most popular actor implementations is Akka [185]. Akka is intended for building highly concurrent, distributed, and resilient message-driven applications in Java and Scala. It is widely considered *the* implementation of the actor model on the JVM. Akka is said to handle up to 50 million messages per second and cites a memory footprint of 2.5 million actors per GB of heap[2]. We examine how reactors in C and C++—our most mature targets—stack up against Akka actors. All obvious differences aside (reactors vs. actors, compiled languages vs. JVM-based), the point of this comparison is to see whether reactors and actors can play in the same league. The question is whether the cost of synchronization in reactors is acceptable or prohibitive compared to state-of-the-art frameworks for building concurrent software. Can we realistically have performance and determinism, too?

The default Akka configuration that we compare against automatically chooses an optimal number of worker threads based on `Runtime.getRuntime().availableProcessors()`, let us call it N. On our evaluation system, $N = 24$. In both the C and C++ runtime, we have found no advantage to using a number of threads greater than N. The number of worker threads in an LF program can be specified using a target property (see Section 3.2).

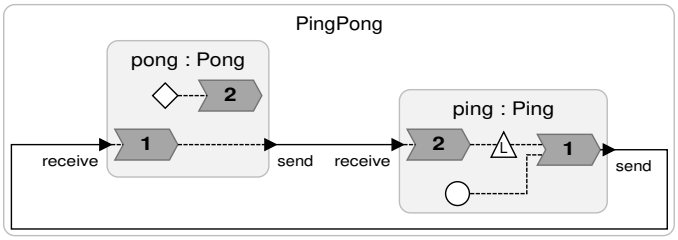


Figure 4.9: A reactor implementation of the Savina PingPong benchmark.

Micro Benchmark: Ping Pong

The first benchmark we discuss is the "Hello World" equivalent of an actor program. One actor sends a message to a receiving actor that simply returns the message to the sender. This sequence gets repeated many times, and the faster the program completes, the more efficient the runtime system is. In other words, this benchmark provides an indication of the overhead induced by the runtime system. A diagram of the reactor implementation of the PingPong benchmark is shown in Figure 4.9 and the benchmark results are shown in Figure 4.10.

Because there is no exploitable parallelism in this benchmark, we gain no benefit from using more than one thread. Our single-threaded runtime, which has no dependency on pthreads and is therefore more suitable for bare-metal embedded platforms, runs this benchmark a bit faster than our multi-threaded run times—it has less overhead. Our single-threaded C runtime is consistently more than 25× faster than Akka for this benchmark, but even the multi-threaded reactor runtimes beat Akka by an order of magnitude.

[2] `https://akka.io/`

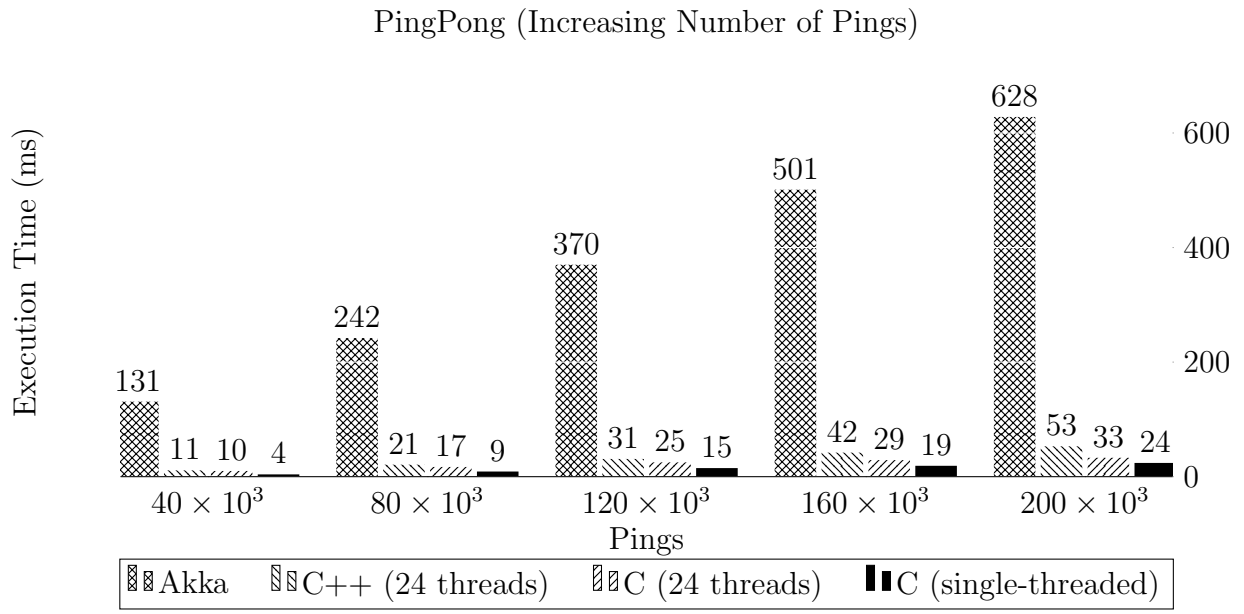


Figure 4.10: PingPong: a comparison between Akka actors and reactors.

Concurrency Benchmark: Dining Philosophers

This benchmark is based on the classic concurrency problem where a group of monks, sitting around a round table, alternate between thinking and eating noodles. In order to eat, each monk needs two chopsticks. Each adjacent pair of philosophers shares access to a single chopstick that they can acquire or release. This problem was originally formulated in 1965 by Edsger Dijkstra and was given its present formulation by Tony Hoare [96]. The problem captures the basic principle of mutual exclusion and cleverly illustrates the problems of deadlock and starvation. The solution implemented in this benchmark uses an arbitrator that instructs philosophers what to do.

Our implementation of this benchmark revealed a deficiency in our threaded runtime that we are still in the process of addressing. The root of the issue is that when new reactions are pushed onto the reaction queue, worker threads are notified and compete for the work, which leads to a lot of contention and no meaningful exploitation of parallelism because the reactions of the philosophers take very few cycles. Initial results show that reducing the amount of signalling or letting idle workers poll the reaction queue after a timed wait significantly reduces contention and leads to performance comparable to Akka when it comes to this particular benchmark. However, to understand the ramifications of such change for other types of workloads, more investigation (and the implementation of more benchmarks) is necessary.

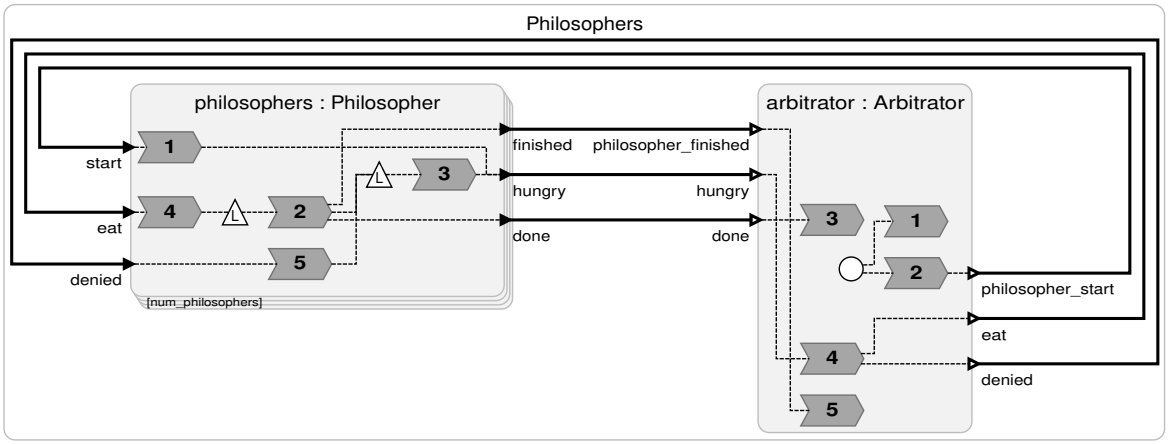


Figure 4.11: A reactor implementation of the Savina Philosophers benchmark.

Parallelism Benchmark: Trapezoidal Approximation

The third and last benchmark we discuss concerns a typical master-worker pattern in which a master process divides a problem into several sub-problems and tasks workers to solve them. The task at hand in this particular benchmark is to approximate the area of a trapezoid. The reactor implementation of this benchmark is depicted in Figure 4.12.

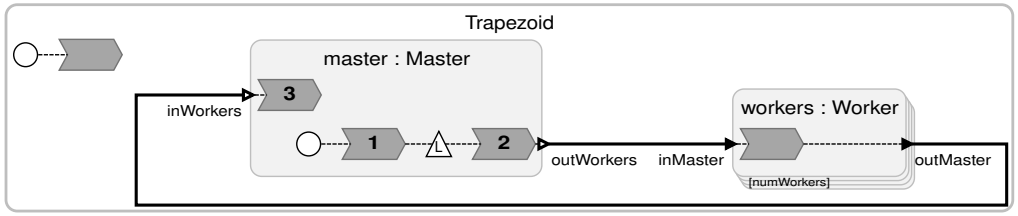


Figure 4.12: A reactor implementation of the Savina Trapezoid benchmark.

Let us first examine the ability of the reactor runtime to exploit parallelism among the workers. For that to occur, multiple threads are needed. We expect the execution time of the program to scale down with the number of threads, as long as there are independent cores to map those threads to. Our evaluation system has 6 physical cores and 24 hardware threads. As shown in Figure 4.13, we see a close-to-linear speedup with the number of threads up to 6 threads. Beyond that point, adding more threads still reduces execution time. Overall, we see logarithmic curve that clearly flattens out around 24 threads. The performance between C and C++ is similar.

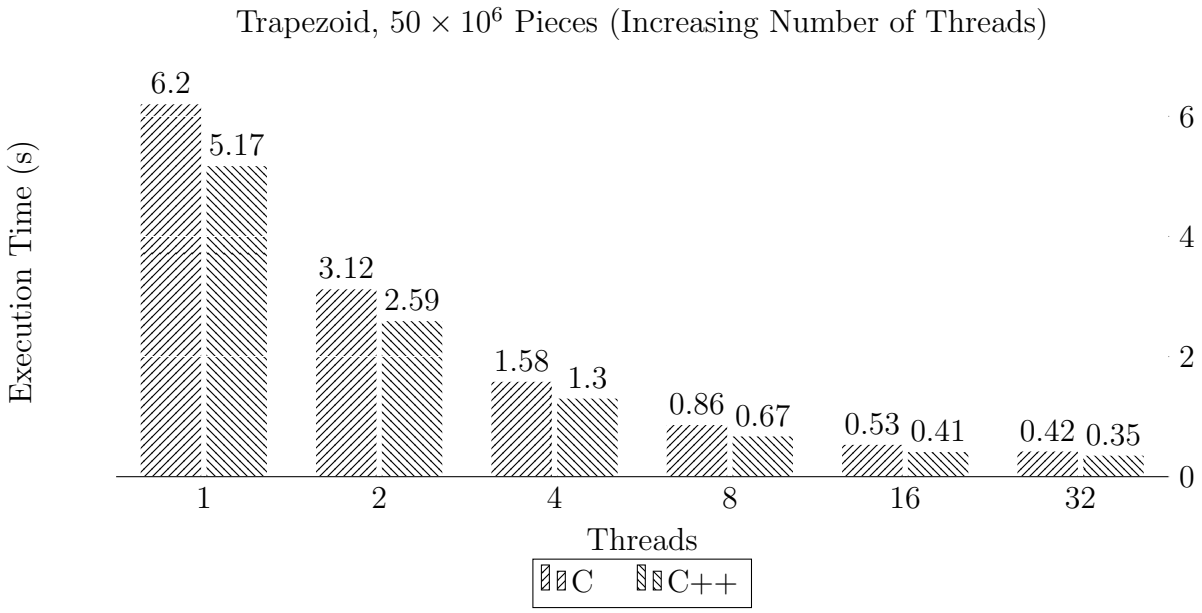


Figure 4.13: Trapezoid: reduced execution time with a larger number of worker threads.

When comparing against Akka (see Figure 4.14), reactors again come out on top. The difference is less dramatic with this benchmark, but the C runtime outperforms Akka by a factor 2.3, and C++ is 2.85× faster. While a case can be made that the PingPong benchmark is not representative of a useful program or meaningful workload, the master-worker pattern certainly is, and the performance of reactors does not disappoint. While it is premature to say that reactors can compete with actors on *all* fronts, we have established that reactors are at the very least competitive on *some* fronts. More work is needed to fully understand the strengths and weaknesses of reactors, but their ability to outperform a state-of-the-art actor framework like Akka, is promising.

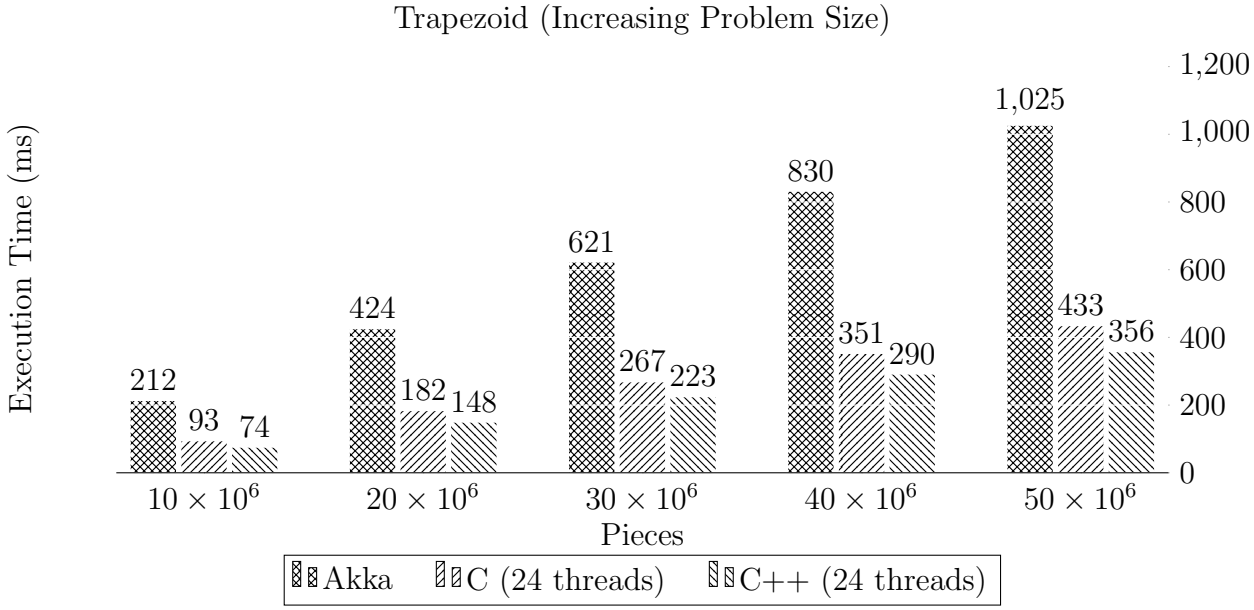


Figure 4.14: Trapezoid: a comparison between Akka actors and reactors.

The distinction between the past, present and future is only a stubbornly persistent illusion.

Albert Einstein

Chapter 5

Federated Execution

This chapter draws from and expands on previously published work titled "A Language for Deterministic Coordination Across Multiple Timelines" [143] that was co-authored with Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A. Lee.

Actors [94, 2], as realized in Erlang [9], Akka [185], and Ray [168], are commonly used for building distributed software, where each actor could potentially reside on a different node and exchange messages with other actors via a network. Reactors are also suitable for this. We refer to a reactor of which contained reactors are mapped to individual process that exchange messages as a **federated reactor**, or simply a **federation**. We call each reactor in a federation that gets maps to its own process a **federate**.

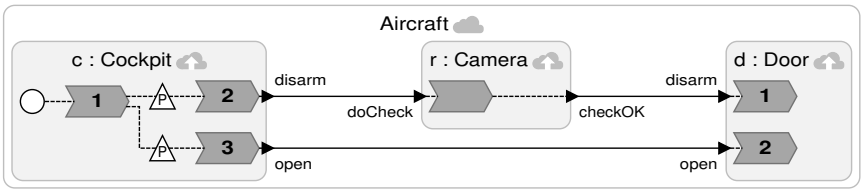


Figure 5.1: A federated reactor that controls an aircraft door. Each reactor runs on a different host.

Let us consider a federated version of the nondeterministic actor program discussed in Chapter 1, depicted in Figure 5.1. In this application, we suppose that a commercial aircraft manufacturer wishes to automate the opening of an aircraft door. The Cockpit reactor responds to a button press in the cockpit and sets its two outputs disarm and open. The Camera reactor performs a visual check to confirm whether a ramp is present outside the aircraft. Only if a ramp is present, the Camera sets its checkOK output to `true`, causing the disarm input of the Door to be present. The Door reactor, hosted on a networked software component residing in the aircraft door, has two inputs: disarm and open. An event on the

disarm input triggers a reaction that disables deployment of emergency escape slides if the door is armed. A second reaction in the Door, triggered by the open input, opens the door. If the door is opened when it is armed, then the slides will deploy.

In a federated execution, assigning a value to a port translates into a message being sent over the network. Using a protocol with reliable in-order message delivery (e.g., TCP), we can assume that messages sent between any of the reactors arrive in the correct order with respect to other messages *originating from the same sender*, but for the Door reactor it is critical that messages from different senders (i.e., the Cockpit and Camera reactor) are observed in the correct order. A failure to satisfy this constraint could lead to an unintended emergency slide deployment, which is both dangerous and costly.

To ensure determinism in a federated program, it is essential to preserve tags across networked communication. For this, it is necessary to transmit tags along with the messages. A more subtle issue is that a federate must avoid advancing logical time ahead of the tags of messages it has not yet seen. This problem has many possible solutions, many of them realized in simulation tools [75]. However, LF is not a simulation but an implementation language, which introduces unique problems. In this chapter we discuss how federated execution is realized in LF.

5.1 Reasoning About Time

It is impossible, from first principles in physics, to determine the order in which two geographically separated events occur. There is no such thing in physics as the "true" order in which separated events occur. There is only the order seen by an observer, and two observers may see different orders. Hence, it would be an unrealistic goal to require that if a disarm message is "truly" sent before an open message, then the door will be disarmed before it is opened. To use such a requirement, we would have to identify the observer that determines the outcome of the predicate "before."

One choice of observer, of course, is the receiver of the messages, the microprocessor in the door that performs the disarm and open services. This is the choice made in an actor model, (as well as publish-and-subscribe and service-oriented models), but as we have shown, it leads to clearly undesirable outcomes. Even if the disarm and open messages originate from the same source, they may arrive out of order. The originator sees a different order from the recipient, as shown in Figure 5.2.

Only if, instead of relying on a physical notion of time, we define a *logical* or *semantic* notion of time, does it become possible to ensure that every observer sees events in the same order. This will require a careful definition of "time" as a semantic property of programs. We will also have to stop pretending that our logical notion of time *is* physical time, and instead accept a multiplicity of observers and understand the relationships between their timelines.

Sidebar: Distributed Discrete Event Models

Discrete-event models of computation, where time-stamped events are processed in timestamp order, have been used for simulation for a long time [221, 41]. 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 [44] 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 [101], 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.

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.

A third approach is High Level Architecture (HLA), which is a standard for distributed simulation in which several simulations can interact through a message-oriented middleware layer called a Run-time Infrastructure (RTI). This middleware provides services for message exchange, synchronization, and federation management. The standard was developed in the 90s under the leadership of the US Department of Defense [53] and was later transitioned to become an open international IEEE standard. Some of the terminology we use to describe entities in the distributed execution of reactors is borrowed from HLA, including the notion of "federates" and an entity called RTI.

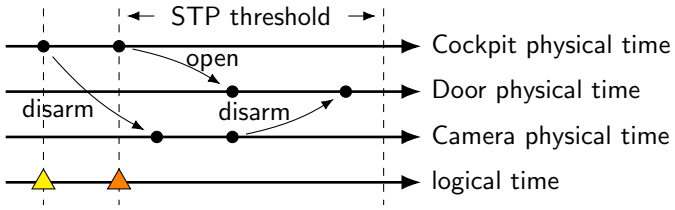


Figure 5.2: Different observers may see events in a different order. An additional logical timeline allows to establish a global ordering. After a certain safe-to-process (STP) threshold, Door received all relevant messages and can use the logical timeline to determine that disarm should be processed *before* open.

One way to provide a semantic notion of time is to use numerical timestamps [118]. If messages carry timestamps, then our requirement can be that every federate processes messages in timestamp order. If we further require that messages with identical timestamps be processed in a predefined deterministic order—as reactors do—then our semantics will ensure that any two reactors with access to the same messages will agree on their order. We know from experience with distributed discrete-event simulators, however, that it is challenging in a distributed system to preserve timestamp order [75]. Moreover, here, we are not interested in *simulation*. We are interested in cyber-physical *execution*, where physical time and (imperfect) measurements of physical time play an important role. The methods used for distributed simulation will have to be adapted, as we do here.

The use of timestamps superimposes on our distributed system a logical timeline that must coexist with a multiplicity of timelines, measurements of physical time, and with actual physical time. Timestamps must originate somewhere. In reactors, the scheduling of physical actions facilitates the creation of events with tags based on physical clocks, and those same physical clocks lend a rigorous meaning to deadlines with respect to the processing of events with a certain tag. As we will see, these building blocks can be used to preserve the deterministic execution semantics of reactors also in federated reactor programs. Unlike in untimed systems, it is detectable when determinism is lost; soon as a situation occurs where a federate has moved its execution beyond the tag of an incoming message, it is clear a fault must have occurred. This detectability enables the design of **fault-tolerant systems**.

The logical timeline together with the requirement that messages be processed in timestamp order provides a *model* of our system. Of course, no physical realization of a system can be assured of always behaving like its models. Even the most carefully designed silicon chip, for example, may violate the behavior of the logic diagram that defines its design. Every engineered system will behave correctly only under some assumptions. The assumptions for a silicon chip, for example, may include a temperature range. The approach we give here has the distinct advantage that our assumptions are explicit and quantified.

In the aircraft door example, we can employ a decentralized coordination scheme to ensure a system behavior that is repeatable, in that, given the same timestamped inputs,

the response will always be the same. This solution requires that when the Door federate receives a open message with tag g, it waits until its local physical clock hits a precomputed threshold before acting on that message (cf. Figure 5.2). This will allows Door to continue listing for other messages with a tag that is earlier or equal g and handle those prior or simultaneously with the open message that it received. This guarantees that the open message will be handled in timestamp order relative to other messages, including any disarm messages that may originate anywhere in the system. The assumptions will include a bound E on the clock synchronization error, a bound L on the network latency, and a bound X on the execution time of certain pieces of code. What bounds are acceptable is application dependent. Existing technologies can let us tighten bounds on E [100], L [113], and X [225, 187].

In reality, *any* reasonable handling of an open message has to make these same assumptions. If there really is no bound on network latency, how can we possibly reason about the order in which messages are handled? If clocks differ wildly across a distributed system, how can we expect any coherent notion of "before"? In LF, these assumptions can be made explicit, quantified, and their violation detectable.

5.2 Decentralized Coordination

In a coordination approach based on Ptides [223], which we call **decentralized coordination**, it is a requirement that the physical clocks on all federates be synchronized with some bounded error, using for example NTP [163], IEEE 1588 [67], or HUYGENS [82]. Synchronizing physical clocks enables decentralized, fault-tolerant, and bottleneck-free federated execution while preserving the semantics of logical time. Ptides also requires being able to bound network latencies and (certain) execution times. These three bounds (clock synchronization error, network latencies, and certain execution times) have to be made explicit. The technique used by Ptides has been shown to scale to very large systems; it is used in Google Spanner, a global database system that coordinates thousands of servers [50].

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. That is, if you simultaneously ask two nodes what time it is, they will not disagree by more than E. Second, they assume that every network connection between nodes has a bound L on the latency for message delivery. This assumption is necessary anyway for many realtime applications.

These two assumptions, E and L, may, of course, be violated in any physical deployment of a physical system. Hardware failures or malicious attacks, for example, could cause violations. One interesting property of reactors is that such violations are detectable. They result in out-of-order timestamps. This condition can be detected at run time as a fault condition, enabling fault-tolerant system designs that adjust themselves to such fault conditions. Moreover, the assumptions E and L are explicit and quantified. Many practical system designs

make such assumptions *implicitly* and without quantification, making detection of violations difficult.

Example: A Distributed Database

We can use Spanner's database application to explain how these two assumptions enable efficient and deterministic federated execution. Consider a distributed database for a reservation system, where the data is replicated on two different platforms, PlatformA, depicted in Figure 5.3 and PlatformB, depicted in Figure 5.4. Assume that the two copies of the database are initially identical and that an update query arrives through WebServerA on PlatformA that makes a change to a record in the database. Queries to the database will be tagged, and the correct response of the database will be defined by the numerical order of these tags.

At the logical start time of the execution, the first reaction of WebServerA 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, say an update to a record to make a reservation, the SCHEDULE procedure is invoked to schedule an event for its physical action, which is a trigger for the second reaction of WebServerA. The tag g_u of the scheduled event is obtained from the local physical clock, and the second reaction will execute at a logical time equal to g_u. The second reaction will forward the tagged message to DatabaseA, which then publishes via NetworkSender the update to all other replicas of the database, including DatabaseB. The dissemination of the update incurs

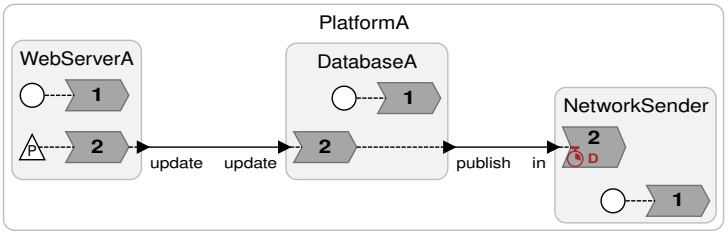


Figure 5.3: Webserver that receives updates, stores them in a local database, and forwards them to are remote database.

network latency that is assumed to not exceed some quantity L. Furthermore, we know that the physical time at which the update arrives at NetworkSender cannot exceed D due to a deadline that is attached to it, indicated by the small red clock symbol the second reaction of NetworkSender. Hence, the event will arrive at PlatformB before physical time on PlatformA exceeds $\pi_1(g_u) + D + L$. Because of clock synchronization error, this event will arrive at PlatformB before physical time *as measured on PlatformB* exceeds $\pi_1(g_u) + D + L + E$.

At around that same time that PlatformA receives the update query, suppose that PlatformB receives a query for the value of the same record being updated at PlatformA. How should the system respond? In Spanner (and Ptides), this query at PlatformB will also be

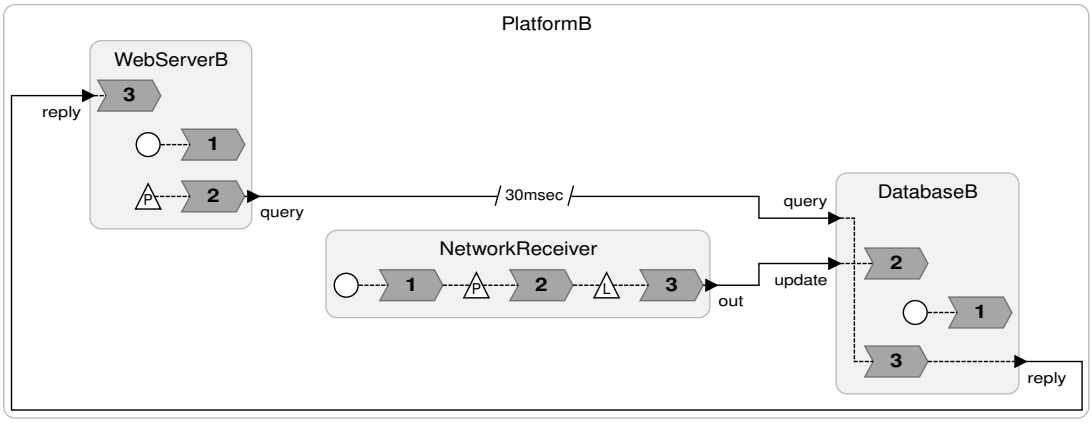


Figure 5.4: Webserver that receives queries, forwards them to a local database, and serves a reply.

tagged using the local physical clock, and the semantics of the system defines the correct response to depend on the numerical order of the tags of the two queries. If the query at PlatformA has an earlier or equal tag to that at PlatformB, then the correct response is the updated record value. Otherwise, the correct response is the value before the update.

Suppose that DatabaseB has a query with tag g_q coming from WebServerB. Can it safely respond to that query? To be safe, it has to be sure that it will not receive an event via its NetworkReceiver with a tag smaller than or equal to g_q after having started processing the event with tagged g_q. How can it be sure?

Such a distributed system could use the Chandy and Misra approach, which would require PlatformA to periodically send tagged null messages to PlatformB. Then, DatabaseB will repeatedly receive null messages on its `update` port with steadily increasing tags. As soon as one of those tags exceeds g_q, it can handle the event on its `query` port that has tag g_q 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 to hold off processing the `query` message until its measurement of physical time exceeds the **safe-to-process (STP) threshold** equal to $\pi_1(g_q) + D + L + E$. As we previously pointed out, if an update to the database is occurring at PlatformA with tag g_u, that update will be seen on PlatformB by physical time $\pi_1(g_u) + D + L + E$. Hence, when the local physical clock exceeds $\pi_1(g_q) + D + L + E$, the event with tag g_q can be safely processed.

Implementing this mechanism with reactors is straightforward. Upon message receipt of the remote event with tag g_u, NetworkReceiver schedules an event with tag g_r using its physical action. Assuming all the assumptions are met, $\pi_1(g_r) \leq \pi_1(g_u) + D + L + E$. Hence, the second reaction of the NetworkReceiver can use a logical action to schedule an event to occur at $\pi_1(g_u) + D + L + E$, triggering the third reaction of NetworkReceiver which will deliver the update to the local copy of the database. In our example we assume $D + L + E = 30\,\text{ms}$. To ensure that queries are processed in order, PlatformB asserts a logical delay of 30 ms

on the connection from WebserverB.query to DatabaseB.query. Such a logical delay can be specified in an LF program using the **after** keyword, which increments the tag. Hence, to determine whether to process the update first or the query first, DatabaseB is effectively comparing tags $\pi_1(g_u) + 30\,\text{ms}$ and $\pi_1(t_q) + 30\,\text{ms}$.

This 30 ms logical delay will translate into a physical delay that is noticeable by a person interacting with WebserverB. The physical time that the system will wait before it starts processing a read query at PlatformB is bounded above by 30 ms. If a faster response is needed a thus a smaller wait time of, say, 20 ms is necessary, then this translates into an engineering requirement that $D + L + E \leq 20\,\text{ms}$. This provides guidance for selection of processing and networking technology and provides a clear criterion for determining what hardware can correctly execute this system with the timing requirements.

Another clear advantage of this approach is that reads to the database generate no network traffic. Only writes that update records generate network traffic.

If any of the assumptions D (the sum of the worst-case execution times of two reactions), L (the network latency bound), or E (the bound on the clock synchronization error) is violated, then the NetworkReceiver may find that $\pi_1(g_r) > \pi_1(g_u) + D + L + E$. At that point, the NetworkReceiver can raise an alarm indicating a fault condition. For a database system, a reasonable reaction to such a fault condition is to reject a transaction. Standard techniques for distributed consensus can be used to accomplish this, but then the overhead incurred by such techniques is rarely incurred. Moreover, the rarity of the occurrence can be controlled by standard engineering methods. But, of course, faults cannot be made impossible.

In this example, we force a federate to observe an STP threshold by inserting logical delays along connections. To achieve the desired behavior, the delay on the connection between WebserverB.query and DatabaseB.query has to match the $\text{STP} = D + L + E$ that is used in NetworkReceiver to adjust the timestamps of the events coming from PlatformA. Alternatively, we could also choose to parameterize each federate f with a threshold STP_f that it then uses to adjust the release time of *all* events it handles. Specifically, on Line 15 of NEXT (see Section 2.7) each f would not wait until $T \geq \pi_1(g_{\text{next}})$, but until $T \geq \pi_1(g_{\text{next}}) + \text{STP}_f$. This would let us preserve the original timestamp of the events coming from PlatformA, but it would also force the reactions to events on PlatformB to be delayed with respect to physical time, potentially causing a noticeable delay in the handling of physical actions in PlatformB.

A trade-off can be made where a portion of the safe-to-process time is absorbed by logical time delays along connections between federates, and the remainder translates into federates imposing extra lag on their handling of events. These choices are ultimately application dependent. Conceptually, Ptides achieves determinism by making the latency in the entire system uniform. This comes at the cost of added latency along paths through the system that are faster than the slowest one. The only way to bring down this cost is to reduce D, L, and E.

Example: The Aircraft Door

Decentralized coordination can also be used in the aircraft door control system in Figure 5.1. In that example, the messages are all logically simultaneous (they bear the same tag), even though the three federates are distributed across different hosts. When the Door federate d receives a message with tag g_m destined for its open port, then it should not invoke the reaction triggered by open until the local clock exceeds $\pi_1(g_m) + \max(D_1, D_2) + E + L$, where D_1 and D_2 are the deadlines associated with the two network interfaces that send messages from the Cockpit and Camera federate. The use of hierarchy ensures that there is a software entity, the container for the three reactors, 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. If bounds on execution times are derivable from the code [216], then D_1 and D_2 can also be derived automatically. Or the system could be realized using PRET machines [130], in which case extremely high confidence in the bounds on the execution times becomes achievable.

5.3 Centralized Coordination

It is not always feasible to obtain (or successfully estimate) reasonable bounds on execution time, network latency, and clock synchronization error. A simpler coordination approach that can be employed that uses a centralized controller called an RTI (Run Time Infrastructure). This approach, which we call **centralized coordination**, is similar to several tools that implement the HLA standard (High Level Architecture) [114]. In this approach, each federate has two key responsibilities:

1. it must consult with the RTI before advancing logical time; and
2. it must inform the RTI of the earliest logical time at which it may send a message over the network.

This centralized approach, however, has three key disadvantages. First, the RTI can become a bottleneck for performance since all messages (except for those that travel through physical connections), must flow through it. Second, the RTI is a single point of failure. Third, if a physical action can trigger an outgoing network message, then the earliest next event time is never larger than the time of the physical clock. This can lead to slow advancement of logical time with many messages exchanged with the RTI.

5.4 Support for Federated Programs in LF

It is possible to convert an ordinary LF program into a federated program simply by substituting the `main` modifier with the `federated` keyword. This effectively turns each reactor instance in the top-level reactor into a federate. Each federate can be mapped to particular

host. In a federated LF program, some parts of the orchestration discussed in the distributed database example are automated. Connections between federates (reactor instances directly contained by a federated reactor) are automatically transformed into entities similar to the NetworkSender and NetworkReceiver reactors in Figures 5.3 and 5.4. A federated version version of the distributed database example is shown in Figure 5.5.

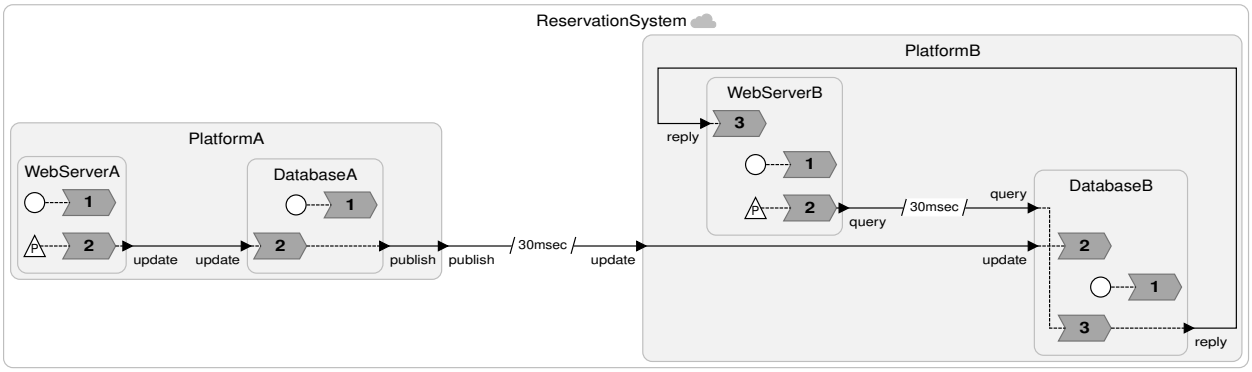


Figure 5.5: A federated LF program with decentralized coordination for a reservation system.

In a **federated execution**, each federate runs in a separate process, potentially on a different machine. If there are n federates in a program, then the code generator will generate $n+1$ separate programs; one for each federate and one for the RTI. Each of these programs is transferred to and compiled on its designated host. A federated program is started by starting the RTI along with all of its constituent federates.

Example Consider the federated program in Listing 5.1. This is a particularly simple form of a federation in which a `Print` federate receives timestamped messages from a `Count` federate. The **`federated`** keyword tells the code generator that the program is to be split into several distinct programs, one for each top level reactor, and one for the RTI. If the filename that contains the code of Listing 5.1 is named `DistributedCount.lf`, then the following three programs will appear in the bin directory:

- `DistributedCount_RTI`;
- `DistributedCount_count`; and
- `DistributedCount_print`.

The root name, `DistributedCount`, is the name of the `.lf` file from which these are generated. The suffixes `_count` and `_print` come from the names of the top-level instances. There will always be one federate for each top-level reactor instance.

In addition, one or two bash shell scripts will be generated:

- `DistributedCount`; and

- `DistributedCount_distributor.sh`.

The first of these is a shell script that launches the RTI and each federate program. The second script, `DistributedCount_distributor.sh`, will be generated if any of the three programs are specified to be run on a remote machine. That script will copy each source file to its prescribed location using `scp` and compile it there (via `ssh`). The program in Listing 5.1 specifies that the RTI shall compile and execute at `rti.lf-lang.org`, whereas federate `count` shall be mapped to `fed.lf-lang.org`, and federate `print` is to compile and run on `localhost`. A prerequisite is that `user` is an existing user on `rti.lf-lang.org` and the system on which the LF program is compiled has to be in possession of a valid private key in order to authenticate. Since no user is specified for `fed.lf-lang.org` the remote username defaults to the name of the local user.

Listing 5.1: Minimal example of a federated LF program under centralized coordination

```
1 target C {coordination: centralized};
2 import Count from "Count.lf";
3 import Print from "Print.lf";
4 federated reactor DistributedCount at user@rti.lf-lang.org {
5     count = new Count() at fed.lf-lang.org;
6     print = new Print() at localhost;
7     count.out -> print.in;
8 }
```

The coordination strategies discussed in Sections 5.3 and 5.3 are both supported in LF. The target property `coordination` can be used to specify which strategy to use; `centralized` is the default. It should be noted that support for federated reactors is currently experimental and still a work in progress.[1]

While it might be possible to carry out a fully distributed start and end of execution, our current implementation of decentralized coordination still uses a central coordinator for that. The issues surrounding the start and end of execution of a federation of reactors covered in Sections 5.4 and 5.4 are identical for both mechanisms. In a federation with decentralized coordination, no communication with the RTI is necessary during execution; each federate independently advances time and reorders incoming messages according to their tags.

Coordinating the Start of Execution

At the start of a federated program, each federate registers with the RTI. When all expected federates have registered, the RTI broadcasts to the federates g_{start}, the logical time at which they should start execution. Hence, all federates start at the same logical time, which is determined as follows. When each federate starts executing, it sends its current physical time (drawn from its real-time clock) to the RTI. When the RTI has heard from all the

[1]Soroush Bateni and Edward A. Lee have been the main developers of the runtime support for federated execution currently present in LF.

federates, it chooses the largest of these physical times, adds a fixed offset (currently one second), and broadcasts the resulting time to each federate.

When a federate receives the starting time from the RTI, unless it is running in `fast` mode (see Section 3.2), it will wait until its local physical clock matches or exceeds that starting time. Thus, to the extent that the machines have synchronized clocks, the federates will all start executing at roughly the same physical time, a physical time close to g_{start}. If any one of the hosts has a physical clock that is far ahead or far behind the others, then unexpected stalls at startup could result. Hence, a federation should be run only on machines that have some level of clock synchronization, at least, for example, using NTP [163].

Coordination During Execution

When one federate sends data to another, by default, the tag at the receiver will match the tag at the sender. We can also modify the tag by imposing a logical delay on the connection using an **`after`** clause. For connections between federates that are marked *physical* (using the `~>` syntax), the received events are tagged based on a reading of the physical clock of the receiving federate. Even in a centralized federation, the data transmission for physical connections can be done directly between federates instead of through the RTI.

The preservation of tags for events that are conveyed via logical connections between federates implies some constraints—even under centralized coordination. We already know that the presence of a realtime reactor (see Section 3.5) in a federate precludes the federate from advancing its logical time past the current reading of its physical clock. Let us conservatively assume that this constraint applies to *all* federates. This means that an event with tag (t, m) cannot be injected into the network for transport from some federate A to another federate B until $T_A \geq t$. Consequently, the message from A reaches B after a **physical time delay** bounded by L, the maximum time it takes the message to traverse the network, and E, the clock synchronization error between A and B. In a centrally coordinated federation, the RTI will deny any requests from B to advance time beyond (t, m), for the entire duration that the message from A is in flight. Just like in the distributed database example we discussed in Section 5.2, this can lead to a physical time delay in handling of events that originate from physical actions in B. The cure to this problem is the same we saw in Section 5.2: if the lag induced in B is not acceptable, then a logical delay D can be added to the connection between A and B. Provided that $D > L + E$, this means that B will no longer be forced to lag behind its physical clock due to messages coming from A.

Related to this issue, but more problematic, is the following. Suppose federates A and B are put in a feedback loop, where A receives messages from B, and B receives messages from A. This configuration does not only let messages from A to induce lag in B; it also allows messages from B to induce lag in A. This type of interaction could lead to a divergence of the amount by which A and B each lag behind their physical clock. Specifically, this is possible when $2(L + E) - (D_{BA} + D_{AB}) > 0$, where D_{AB} denotes the logical delay on the connection from A to B and D_{BA} is the logical delay on the connection from B to A. In other words, unless the logical delays in the federated program mask the physical delays

in its realization, the approximate synchronization to physical time may be lost. This is not a very surprising observation for those familiar with the LET paradigm [108], but it is a phenomenon not ordinarily observed distributed systems, which tend to either rely on a purely logical notion of time (e.g., Lamport clocks [118]), or only consider physical time.

Coordinating the End of Execution

A federated execution can come to a halt for several reasons:

- Starvation: there are no more events on the event queue;
- Timeout: there is a predefined g_{end}, an upper bound on the tag of the last event;
- Requested stop: reaction code has requested a stop (see Algorithm 3); or
- External signal: Execution is terminated externally with Control+C or `kill`.

These situations are covered in Section 2.7 for non-federated reactors, but in a federated context there are a number of subtleties that are worth discussing.

Starvation

When a federate has an empty event queue (and target property `keepalive` is not set to `true`), then the federate cannot simply invoke the SHUTDOWN procedure (see Algorithm 14), because other federates might supply it with future work. Only when *all* federates are starved, the federated execution can conclude. While there are many possible solutions for solving this distributed consensus problem, we currently solve it by letting each starving federate report to the RTI the total number of messages it has sent or received on each direct connection it has to another federate. When the RTI has received such a message from all federates, and the number of messages sent and received on each direct connection matches, RTI broadcast a shutdown message.

Timeout

The target property `timeout` specifies g_{stop}, the last logical time at which reactions should be triggered, computed relative to g_{start}. Just like in an ordinary non-federated program, shutdown reactions will execute at g_{stop}, along with whatever reactions might be triggered by events that are scheduled to happen at g_{stop}. One noteworthy subtlety is that events conveyed through a physical connection are likely to get lost if they occur near g_{stop}. This is simply because those events get (re)tagged based on physical time. If the assigned tag is greater than g_{stop}, then the event will not be handled (just like any other events with a tag greater than g_{stop} that might be present in the reaction queue).

Requested Stop

When a reaction inside a federate invokes requestStop (see Algorithm 3 in Section 2.5), then all federates have to come to agreement as to what the last tag g_{stop} should be—a similar consensus problem as the determination of g_{start}. Upon receiving a shutdown request, the RTI asks each federate to report the earliest future tag at which it can execute the normal shutdown sequence during which all reactions triggered by $\diamond$ are executed. The RTI then picks the largest tag and tells all federates to set their g_{stop} accordingly.

External Signal

Each federate and the RTI should catch external signals to shut down in an orderly fashion. When a federate gets such an external signal (e.g., control-C), it should inform the RTI that it is resigning and write an EOF (end of file) to each of its socket connections to other federates. The RTI and all other federates should continue running until some other termination condition occurs. When the RTI gets an external signal, then it should act as if a stop was requested by one of the federates. This means finding the first possible g_{stop} and executing the normal shutdown sequence.

5.5 Conclusion

We have shown that the deterministic semantics of reactors can be preserved even when reactors are mapped to separate processes and distributed across hosts, either using a centralized coordination scheme modeled after HLA [114], or a decentralized coordination scheme based on Ptides [223]. In either of these schemes, time-related subtleties arise. In the decentralized case, determinism can only be preserved under well-stated assumptions about bounded execution times, network latency, and clock synchronization error. While centralized coordination does not require explicit bounds on physical time delays in order to guarantee a deterministic ordering of events, ignoring physical time delays can still lead to adverse (and unexpected) system behavior. In either scheme, physical time delays due to processing, message transport, or clock skew, can cause one federate to prevent another federate from advancing time and handling local events in a timely manner. Under centralized coordination, circumstances exist where feedback between federates can even lead to a divergence between logical time and physical time.

We argue that these kinds of problems are structural, quantifiable, easy to diagnose, and straightforward to address. One solution is to add logical delays in the software to accommodate physical delays in the realization; another is to make the physical realization faster and more time-predictable. If the cost of determinism in terms of the required latency is too high—or the application simply does not require determinism—physical connections can be used to remove message-ordering guarantees (and the scheduling constraints imposed by them). One can think of the receiving end of a physical connection as a simpler version of the NetworkReceiver in Figure 5.4 that makes received messages available directly (using a phys-

ical action) rather than by means of a logical action in observance of some safe-to-process threshold. 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 tag 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 tags as part of the test vector.

Adopting this approach to engineering distributed systems, however, requires a reckoning with the fact that we can no longer dismiss time as a mere metric for performance. It will require a paradigm shift in the thinking of engineers, and an investment in technologies that can drive down the cost of determinism in terms of the latency that it requires. This includes low-latency and high-bandwidth networking technology, clock synchronization mechanisms [141], and processors for which tight bounds on execution time of reactions can be computed [122].

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan M. Turing

Chapter 6

Conclusion

6.1 Further Work

This work opens up a many avenues for further work. Let us discuss some of them.

Performance Analysis

The preliminary benchmarking efforts discussed in Section 4.5 show promising results, but more work is needed to fully understand the strengths and weaknesses of reactors when compared to actors. We expect that implementing the remaining benchmarks from the Savina suite [99] will provide a fuller picture, as well as opportunities for improving LF and its runtime implementations. We have only recently started to develop tracing capabilities for our C and C++ runtime implementations, which will certainly be an important aid in diagnosing performance bottlenecks. It has already revealed that scheduling policies great affect performance. Further work could focus on exploring trade-offs in the scheduling of reactions, and finding methods for tuning the runtime scheduler to different types of workloads.

The runtime support for federated execution of reactors (Chapter 5) is still in a relatively early stage of development and has not yet been subjected to any performance evaluations. It would be interesting to see how federated reactors would stack up against well-established message passing frameworks like MPI [85] or actor-based frameworks like CAF [45], Ray [168], Akka [185], or (Scalable) Distributed Erlang [46].

While benchmarks are important indicators, they seldom serve as predictors of how well a language or framework fares in practice. The implementation of good demonstrator applications would certainly help prove the viability of the concepts discussed in this thesis. In pursuit of such proofs of concept, future work could be aimed at augmenting the Robot Operating System (ROS) [181] with a deterministic coordination layer based on LF Currently, ROS relies on a publish-subscribe mechanism for communication between nodes. Another potential ecosystem in which to leverage reactors is Autoware [105]. Robotics and vehicular software aside, demonstrator applications that appeal to the imagination could be sought

in the areas of Virtual Reality (VR) or computer music, which are also reactive and time-sensitive by nature.

Formal Verification

Existing work around verifying actor-based programs has focused on constructing labelled transition systems (LTS) and performing model checking to find execution traces that violate system specifications (e.g., [198]). Dynamic partial-order reduction (DPOR) techniques have also been successfully leveraged for the verification of actor systems [204]. The verification of reactor programs presents a new and open problem. Reactors are intended to operate in cyber-physical systems, which pose a unique challenge to the formulation of verification problems; the idea of "state" that is central to the concept of model checking, has no well-defined meaning in physical reality without involving the notion of an observer [197]. This issue aside, one way to furnish support for verifying properties about reactor programs would be to create an LF target based on a modeling and verification language like UCLID5 [193]. The verification machinery of the target language could then be used to prove or disprove properties about LF programs.

Runtime Improvements

Support for Mutations

Support for mutations in our runtime implementations is still under development. Our TypeScript runtime has (so far) made the most progress toward implementing mutations, and beginnings toward this goal have been made in the C target as well. Several of the Savina benchmarks (such as the Sieve of Erathosthenes) actually require runtime mutations. While we have a working C implementation of some of those benchmarks, they currently require reaching deep into the internals of the runtime library to carry out dynamic reconfiguration tasks. These procedures need to be abstracted and made available through the low-level API that we outline in Section 2.5. In addition to that, it would be desirable to have higher-level API functions for creating common patterns such as fork-join configurations and pipelines.

Preemptive EDF Scheduling

The EDF-based scheduling policy in our C runtime is nonpreemptive, which can lead to deadline misses that are preventable under a policy in which running reactions could be paused in order to free up resources to reactions with earlier deadlines. There are various ways of accomplishing preemption—some being more portable than others. Under a default round-robin time-sharing policy that is common on most existing Linux and Unix platforms, "nice values" can be used to increase the priority of a thread. How exactly those thread priorities affect the scheduling of threads is in the hands of the kernel. It would be interesting to investigate whether dynamically changing nice values (using the pthreads `setpriority` function) could help improve the likelihood of meeting deadlines.

Lock-free Data Structures

The C and C++ runtime implementations rely on mutual exclusion locks to protect shared data structures such as the event queue and reaction queue. Contention on locks can be detrimental to performance, and there may be ways to reduce the reliance on locks by leveraging lock-free data structures [211] that rely on atomic hardware instructions.

Exposing Even More Parallelism

As mentioned in Section 4.4, there are still unexplored opportunities for exploiting more parallelism in the runtime system. These opportunities range from optimizations in the assignment of chain IDs to the relaxation of the barrier synchronization that normally occurs before logical time advances. Aside from the dependencies that are readily exposed in LF programs, timing information (such as offsets and periods of timers, minimum delays and minimum spacing for actions) can also be used to inform optimizations in the runtime scheduler.

Language Improvements

Syntax for Common Patterns

While the explicit connections between reactors enable the dependency analysis required for the execution of reactors, drawing connections between ports on a one-by-one basis can be a tedious programming task. The syntactic constructs for multiports and banks of reactors greatly simplify this task, but only for a subset of useful connection patterns. Matrix-like arrangements, fully-connected connection topologies, or pipelines, for example, are still difficult to express. Similar problems exist in hardware description languages. In VHDL, the `generate` statement allows the digital designer to iteratively replicate and expand logic. A similar mechanism might be suitable for LF.

Finite state machines (FSMs) are commonly used to model control logic. A reactor can implement an FSM using state variables; the concept of modes, transitions, and the behaviors associated with them would be encoded in the bodies of reactions. From a software engineering perspective, it would be helpful to make these concepts visible at the LF level—in the code, and, perhaps even more importantly, in the diagram synthesis. Leading examples of such functionality are SCCharts [89] and the modal models in Ptolemy II [71].

Import and Package System

LF has a simple import system that requires imported classes to be listed explicitly. Name disambiguation must be performed through an aliasing mechanism in the import statement itself; there is no use of fully qualified names. The files in which to locate reactor classes are identified by a (relative) path in the import statement. Whereas imported files are currently looked up relative to the current location of the source file that is being compiled, we plan

to develop a package system that will allow the classpath to be augmented using file-based package descriptions similar to those used in language like Rust and Python.

WCET Analysis in the Compiler

Central to the reactor model is a semantic notion of time that is used to enforce a well-defined ordering of events. LF is suitable for targeting anything from small bare-iron embedded controllers to multi-core shared-memory systems and distributed systems. Having time as a first-class citizen in the language, LF also holds promise as an excellent programming model for real-time systems. But to fully realize this potential, the LF compiler has to be augmented with worst-case execution time analysis capabilities. Inspiration for this could be drawn from Fuhrmann et al. [74]. Given these tools, it should be possible to write LF programs with hard timing guarantees.

Static Schedule Synthesis

The dynamic scheduling of reactors is very flexible. For programs that do not need this flexibility, it might be more appropriate the generate a static schedule. For instance, if a reactor program consists of a network of reactors that abide by the principles of synchronous dataflow [120], then the execution can be performed according to a static SDF schedule synthesized by the compiler instead of the generic reactor runtime scheduler.

Targeting Time-predictable Hardware

In order to realize reactor programs with ironclad timing guarantees [147], reactions must be amenable to WCET analysis, which is necessary to perform schedulability analysis [69, 18]. Platforms that are optimized for predictable timing allow for tighter bounds on WCET and more accurate release times, allowing for better utilization and tighter synchronization to physical time, respectively. GameTime [192, 194], a tool for the timing analysis of software, would be able to achieve much higher accuracy using time-predictable hardware. Two such predictable-time platforms are Patmos [187] and FlexPRET [225].

Patmos is an architecture that is specifically designed to simplify WCET analysis and is supported by several WCET analysis tools. At this time we have already successfully run LF programs on Patmos and have computed WCET for reactions. The multi-threaded C runtime has also been confirmed to run successfully on Patmos with its recently acquired support for pthreads [1]. A closer integration between the LF and Patmos compiler is planned.

The FlexPRET microarchitecture is a realization of a PRET machine [137, 122], which achieves repeatable timing by using a thread-interleaved pipeline, scratchpad memory instead of caches, and a specialized DRAM controller that ensures time-predictable memory access. FlexPRET [225] distinguishes between soft and hard real-time threads, and supports

[1] Thanks to Tórur Biskopstø Strøm at Technical University of Denmark.

an arbitrary interleaving of threads for better utilization given workloads with limited parallelism. We expect that reactors with their explicit timing constraints provide a suitable programming model for PRET machines, which far have been lacking good software support. LF programs specify deadlines, periodic activities driven by timers, and asynchronous external events with constraints on their spacing. The question is how to map reactions onto hardware threads and synthesize the schedules for hard real-time threads so that deadlines are met.

Improving Robustness of Federated Execution

Clock Synchronization

Hardware support for synchronizing clocks (IEEE 1588-2008) [67] is becoming more prevalent, but the effort (and required administrative privileges) involved in setting up Time Sensitive Networking (TSN) could form a barrier to the adoption of federated coordination. Integrating software-based clock synchronization capability (such as HUYGENS [82]) into the federated runtime system, could alleviate this problem.

TSN, on the other hand, offers clock synchronization, flow control, and prioritized routing—capabilities that federated LF programs can take advantage of. Beyond working assumptions about the network into LF code, an interaction between the LF runtime system and TSN configuration could potentially be established.

Handling Late Messages

A message that arrives at a federate bearing a timestamp that is earlier than the federate's current logical time, exposes a fault condition that can occur in a decentralized federation when the assumptions about timing are not met. It means that the chosen STP threshold was too small. An exception like this should probably be handled in an application-specific way, much like deadline misses are. Adding language support for handling these kinds of exceptions would be very useful. A sensible response to receiving late messages could be to increase the STP threshold to reduce the likelihood of receiving late messages in the future.

Detecting Failures

Under decentralized coordination, it is not always detectable when a federate crashes. If an upstream federate stops sending messages this could either be because it has no events, but it could also be because message are getting lost, or because the federate itself could have stopped working. Heartbeat messages [3] could help detect such problems.

Security

Currently, our federated runtime uses a rudimentary form of access control to prevent federates from joining the wrong federation. This should be enhanced to create an encrypted

virtual private network for each federation. This could leverage recent work on distributed authentication and authorization [107].

Dynamically Joining or Leaving A Federation

Mutations could also prove useful in a federated context, where it might be desirable for the number of federates in a federation to change dynamically. To leverage the semantics of mutations for this, it seems necessary to let the RTI be synonymous with top-level reactor rather than act as an external entity that only coordinates the execution. A redesign along these lines would allow a federated reactor to have reactions and mutations (besides contained reactors), which it currently cannot.

Load Balancing

Federates are currently mapped to hosts manually. It would be useful to have runtime support for the automatic distribution of reactors and common parallel computing patterns like MapReduce [55].

6.2 Applications

The work by Menard et al. [162] shows how the federated use of reactors (as explained in Chapter 5) can be used to correct nondeterminism in a real-world application, namely a brake assistant demonstrator application that is provided by the AUTOSAR consortium for their new AUTOSAR AP framework[2]. AUTOSAR AP is an attempt to accommodate the integration of computationally demanding AI-driven control algorithms that are necessary to achieve autonomous driving. While there has been a lot of attention for the robustness of AI components themselves (e.g., the vulnerability of image classifiers to adversarial input [116]), how to confidently integrate such components into safety-critical systems remains a formidable research question. We think that the balance that reactors strike between rigor and flexibility provides a better match for these kinds of complex integration problems than established models like actors, publish-subscribe systems, or shared memory architectures, in which determinism is virtually unattainable, and testing is notoriously hard [204].

Other recent work [217] suggests that there could be a role for reactors in the software for mobile communication systems such as 5G. The challenges in those kinds of system are similar to the AI-based automotive applications in the sense that they are computationally demanding, time sensitive, and highly dynamic.

Another application area in which reactors could prove useful is the realm of Programmable Logical Controllers (PLCs) [191], an old but tenacious technology that has seen little innovation since the late 1980s. While technology trends towards more sophisticated networks, multi-core architectures, and increasingly complex microprocessor architectures,

[2] https://www.autosar.org/working-groups/adaptive-platform/

the specialized programming model used in PLCs makes it difficult to accommodate and take advantage of these technological advances. Reactors could help bridge this gap and drive a new wave of innovation toward increased flexibility that does not compromise the safety guarantees that PLCs are known and praised for.

6.3 Final Remarks

The results of this thesis comprise a formal model of reactors; a description of a polyglot coordination language, compiler toolchain, and runtime system that is capable of delivering determinism in potentially complex and highly concurrent and potentially distributed reactive systems; and a preliminary evaluation that suggests that the deterministic concurrency of reactors does not come at a prohibitive loss in performance. This is a remarkable result because asynchronous and nondeterministic models of concurrent computation (e.g., actors [95], publish-subscribe [160], distributed shared memory [167]) have been pursued and implemented in large part for their efficiency and performance in multi-core and distributed software. The sacrifice of determinacy in these type of systems appears to be commonly accepted as a necessary cost.

The work in this thesis charts a path forward toward testable and understandable concurrency that is *also* highly performant. While more work is necessary to draw a final conclusion on this matter, we have started to lift the veil on what seems to be a false dichotomy between determinism and the ability to effectively exploit parallelism. We have shown, however, that preserving determinism imposes a cost in terms of latency, a trade-off that is brought to the forefront by the relationship that reactors establish between logical time and physical time.

While the emphasis of reactors is on determinacy, asynchrony and nondeterminism can be realized, through the use of physical actions. Our philosophy is that the interactions between software components should be deterministic by default. In the reactor model, any deviation from that default must either constitute an intentional behavior that is allowed explicitly by the programmer, or it must be due to a fault condition that is to be addressed at runtime.

Bibliography

[1] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. Cambridge, MA: MIT Press, 1986.

[2] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. "A foundation for actor computation". In: *Journal of Functional Programming* 7.1 (1997), pp. 1–72.

[3] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. "Heartbeat: A timeout-free failure detector for quiescent reliable communication". In: *International Workshop on Distributed Algorithms*. Springer. 1997, pp. 126–140.

[4] Peter Ahrens, James Demmel, and Hong Diep Nguyen. "Algorithms for Efficient Reproducible Floating Point Summation". In: *ACM Trans. Math. Softw.* 46.3 (July 2020). ISSN: 0098-3500. DOI: 10.1145/3389360. URL: https://doi.org/10.1145/3389360.

[5] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. "Soft Typing with Conditional Types". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 163–173. ISBN: 0897916360. DOI: 10.1145/174675.177847. URL: https://doi.org/10.1145/174675.177847.

[6] S. Alimadadi, A. Mesbah, and K. Pattabiraman. "Understanding Asynchronous Interactions in Full-Stack JavaScript". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. May 2016, pp. 1169–1180. DOI: 10.1145/2884781.2884864.

[7] Charles André. *SyncCharts: A Visual Representation of Reactive Behaviors*. Report RR 95–52. University of Sophia-Antipolis, Apr. 1996.

[8] James R. Armstrong and F. Gail Gray. *VHDL Design Representation and Synthesis*. Second. Prentice-Hall, 2000.

[9] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in Erlang*. Second. Prentice Hall, 1996. ISBN: ISBN 0-13-508301-X.

[10] Cyrille Artho and Armin Biere. "Applying static analysis to large-scale, multi-threaded Java programs". In: *Proceedings 2001 Australian Software Engineering Conference*. IEEE. 2001, pp. 68–75.

[11] Arvind, Rishiyur S. Nikhil, Daniel Rosenband, and Nirav Dave. "High-Level Synthesis: An Essential Ingredient for Designing Complex ASICs". In: *International Conference on Computer Aided Design (ICCAD)*. 2004.

[12] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. "The landscape of parallel computing research: A view from Berkeley". In: (2006).

[13] Y. Bai. "Desynchronization: From Macro-step to Micro-step". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. Oct. 2018, pp. 1–10.

[14] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang Meuter. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (Aug. 2013), 52:1–52:34. ISSN: 0360-0300. DOI: 10.1145/2501654.2501666.

[15] Theodore P Baker and Alan Shaw. "The cyclic executive model and Ada". In: *Real-Time Systems* 1.1 (1989), pp. 7–25.

[16] Henry C Baker Jr and Carl Hewitt. "The incremental garbage collection of processes". In: *ACM Sigplan Notices* 12.8 (1977), pp. 55–59.

[17] Herman Banken, Erik Meijer, and Georgios Gousios. "Debugging data flows in reactive programs". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 752–763.

[18] S. Baruah. "Schedulability analysis of mixed-criticality systems with multiple frequency specifications". In: *2016 International Conference on Embedded Software (EMSOFT)*. 2016, pp. 1–10. DOI: 10.1145/2968478.2968488.

[19] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. *YAML Ain't Markup Language (YAML™) Version 1.2*. Oct. 2009.

[20] Albert Benveniste and Gérard Berry. "The Synchronous Approach to Reactive and Real-Time Systems". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1270–1282.

[21] Albert Benveniste and Paul Le Guernic. "Hybrid Dynamical Systems Theory and the SIGNAL Language". In: *IEEE Tr. on Automatic Control* 35.5 (1990), pp. 525–546.

[22] Jan A Bergstra and Jan Willem Klop. "ACP$_\tau$ a universal axiom system for process specification". In: *Workshop on Algebraic Methods*. Springer. 1987, pp. 445–463.

[23] Gérard Berry. "SCADE: Synchronous design and validation of embedded control software". In: *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 2007, pp. 19–33.

[24] Gerard Berry and Ellen Sentovich. "Multiclock Esterel". In: *Correct Hardware Design and Verification Methods (CHARME)*. Vol. LNCS 2144. Springer-Verlag, 2001.

[25] Gérard Berry and Georges Gonthier. "The ESTEREL synchronous programming language: design, semantics, implementation". In: *Science of Computer Programming* 19.2 (Nov. 1992), pp. 87–152. DOI: 10.1016/0167-6423(92)90005-V.

[26] Gérard Berry and Georges Gonthier. "The Esterel synchronous programming language: Design, semantics, implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152.

[27] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. "Static Scheduling of Multi-rate and Cyclo-Static DSP Applications". In: *Workshop on VLSI Signal Processing*. IEEE Press, 1994.

[28] Rick Bitter, Taqi Mohiuddin, and Matt Nawrocki. *LabVIEW: Advanced programming techniques*. CRC press, 2017.

[29] Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. "A Survey of Active Object Languages". In: *ACM Computing Surveys* 50.5 (2017), 76:1–76:39.

[30] Max Born. "Quantenmechanik der stoßvorgänge". In: *Zeitschrift für Physik* 38.11-12 (1926), pp. 803–827.

[31] Frédéric Boulanger, Christophe Jacquet, Cécile Hardebolle, and Iuliana Prodan. "TESL: A Language for Reconciling Heterogeneous Execution Traces". In: *ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*. Oct. 2014. DOI: 10.1109/MEMCOD.2014.6961849.

[32] Timothy Bourke and A. Sowmya. "Delays in Esterel". In: *SYNCHRON*. Vol. Seminar 09481. Nov. 2009.

[33] Frédéric Boussinot. "Reactive C: An Extension to C to Program Reactive Systems". In: *Software Practice and Experience* 21.4 (Apr. 1991), pp. 401–428. DOI: 10.1002/spe.4380210406.

[34] Frédéric Boussinot and Robert de Simone. "The SL synchronous language". In: *IEEE Tr. on Software Engineering* 22.4 (Apr. 1996), pp. 256–266. DOI: 10.1109/32.491649.

[35] C. Brooks, C. Jerad, H. Kim, E. A. Lee, M. Lohstroh, V. Nouvellet, B. Osyk, and M. Weber. "A Component Architecture for the Internet of Things". In: *Proceedings of the IEEE* 106.9 (Sept. 2018), pp. 1527–1542. ISSN: 0018-9219. DOI: 10.1109/JPROC.2018.2812598.

[36] Christopher Brooks, Chadlia Jerad, Hokeun Kim, Edward A. Lee, Marten Lohstroh, Victor Nouvellet, Beth Osyk, and Matt Weber. "A Component Architecture for the Internet of Things". In: *Proceedings of the IEEE* 106.9 (Sept. 2018), pp. 1527–1542. DOI: 10.1109/JPROC.2018.2812598.

[37] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. "Concurrent programming with revisions and isolation types". In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 2010, pp. 691–707.

[38] Alan Burns and Andy Wellings. *Concurrent and real-time programming in Ada*. Cambridge University Press, 2007.

[39] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. second. Springer, 2005.

[40] Luca Cardelli and Andrew D Gordon. "Mobile ambients". In: *International Conference on Foundations of Software Science and Computation Structure*. Springer. 1998, pp. 140–155.

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

[42] Adam Cataldo, Edward A. Lee, Xiaojun Liu, Eleftherios Matsikoudis, and Haiyang Zheng. "A Constructive Fixed-Point Theorem and the Feedback Semantics of Timed Systems". In: *Workshop on Discrete Event Systems (WODES)*. 2006. URL: http://ptolemy.eecs.berkeley.edu/publications/papers/06/constructive/.

[43] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. "Concurrent software verification with states, events, and deadlocks". In: *Formal Aspects of Computing* 17.4 (2005), pp. 461–483.

[44] K. Mani Chandy and Jayadev Misra. "Distributed simulation: A case study in design and verification of distributed programs". In: *IEEE Trans. on Software Engineering* 5.5 (1979), pp. 440–452.

[45] Dominik Charousset, Raphael Hiesgen, and Thomas C Schmidt. "CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications". In: *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*. 2014, pp. 15–28.

[46] Natalia Chechina, Kenneth MacKenzie, Simon Thompson, Phil Trinder, Olivier Boudeville, Viktória Fördős, Csaba Hoch, Amir Ghaffari, and Mario Moro Hernandez. "Evaluating scalable distributed Erlang for scalability and reliability". In: *IEEE Transactions on Parallel and Distributed Systems* 28.8 (2017), pp. 2244–2257.

[47] Albert MK Cheng. *Real-time systems: scheduling, analysis, and verification*. John Wiley & Sons, 2003.

[48] William Douglas Clinger. "Foundations of actor semantics". In: *AITR-633* (1981).

[49] Paul Moritz Cohn. *Basic algebra: groups, rings and fields*. Springer Science & Business Media, 2012, p. 17.

[50] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. "Spanner: Google's Globally-Distributed Database". In: *ACM Transactions on Computer Systems (TOCS)* 31.8 (2013). DOI: 10.1145/2491245.

[51] Fabio Cremona, Marten Lohstroh, David Broman, Edward A. Lee, Michael Masin, and Stavros Tripakis. "Hybrid co-simulation: it's about time". In: *Software & Systems Modeling* (Nov. 2017). ISSN: 1619-1374. DOI: 10.1007/s10270-017-0633-6.

[52] James B Dabney and Thomas L Harman. *Mastering Simulink*. Pearson, 2004.

[53] Judith S Dahmann, Richard M Fujimoto, and Richard M Weatherly. "The Department of Defense High Level Architecture". In: *Proceedings of the 29th conference on Winter simulation*. 1997, pp. 142–149.

[54] Adam L Davis. "Introduction to Reactive Streams". In: *Reactive Streams in Java*. Springer, 2019, pp. 1–3.

[55] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Sixth Symposium on Operating System Design and Implementation (OSDI)*. USENIX Association, 2004, pp. 137–150.

[56] Julien Deantoni, Frédéric Mallet, and Charles André. "On the Formal Execution of UML and DSL Models". In: *WIP of the 4th International School on Model-Driven Development for Distributed, Realtime, Embedded Systems*. Apr. 2009.

[57] Jack B. Dennis. *First Version Data Flow Procedure Language*. Report MAC TM61. MIT Laboratory for Computer Science, 1974.

[58] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. *P: Safe Asynchronous Event-Driven Programming*. Report. Microsoft Research, Nov. 2012.

[59] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. "P: safe asynchronous event-driven programming". In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 321–332.

[60] Edsger W Dijkstra. "Letters to the editor: go to statement considered harmful". In: *Communications of the ACM* 11.3 (1968), pp. 147–148.

[61] Hui Ding, Can Zheng, Gul Agha, and Lui Sha. "Automated Verification of the Dependability of Object-Oriented Real-Time Systems". In: *2003 The Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. Oct. 2003, pp. 171–171. DOI: 10.1109/WORDS.2003.1267505.

[62] Clara Benac Earle and Lars-Åke Fredlund. "Verification of Timed Erlang Programs Using McErlang". In: *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*. 2012, pp. 251–267.

[63] Stephen Edwards and John Hui. "The Sparse Synchronous Model". In: *2020 Forum for Specification and Design Languages, FDL 2020, Kiel, Germany, September 15-17, 2020*. IEEE, 2020, pp. 1–8.

[64] Stephen A. Edwards. "Verilog". In: *Languages for Digital Embedded Systems*. Boston, MA: Springer US, 2000, pp. 31–54. ISBN: 978-1-4615-4325-1. DOI: 10.1007/978-1-4615-4325-1_3. URL: https://doi.org/10.1007/978-1-4615-4325-1_3.

[65] Stephen A. Edwards and Edward A. Lee. "The case for the precision timed (PRET) machine". In: *DAC '07: Proceedings of the 44th annual conference on Design automation*. San Diego, California: ACM, 2007, pp. 264–265. ISBN: 978-1-59593-627-1. DOI: http://doi.acm.org/10.1145/1278480.1278545.

[66] Stephen A. Edwards and Edward A. Lee. "The Semantics and Execution of a Synchronous Block-Diagram Language". In: *Science of Computer Programming* 48.1 (2003), pp. 21–42. DOI: 10.1016/S0167-6423(02)00096-5.

[67] John C. Eidson. *Measurement, Control, and Communication Using IEEE 1588*. Springer, 2006.

[68] Moritz Eysholdt and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2010, pp. 307–309.

[69] Timo Feld, Alessandro Biondi, Robert I. Davis, Giorgio Buttazzo, and Frank Slomka. "A survey of schedulability analysis techniques for rate-dependent tasks". In: *Journal of Systems and Software* 138 (2018), pp. 100–107. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.12.033. URL: http://www.sciencedirect.com/science/article/pii/S0164121217303102.

[70] Michael Feldman. *Who is using Ada?* SIGAda Education Grop. URL: https://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html#Banking_and_Financial_Systems.

[71] Thomas Huining Feng, Edward A. Lee, Xiaojun Liu, Stavros Tripakis, Haiyang Zheng, and Ye Zhou. "Modal Models". In: *System Design, Modeling, and Simulation using Ptolemy II*. Ed. by Claudius Ptolemaeus. Berkeley, CA: Ptolemy.org, 2014. ISBN: 978-1-304-42106-7. URL: http://ptolemy.org/books/Systems.

[72] Cédric Fournet and Georges Gonthier. "The reflexive CHAM and the join-calculus". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1996, pp. 372–385.

[73] Daniel P Friedman and David Stephen Wise. *The Impact of Applicative Programming on Multiprocessing*. Indiana University, Computer Science Department, 1976.

[74] Insa Fuhrmann, David Broman, Reinhard von Hanxleden, and Alexander Schulz-Rosengarten. "Time for Reactive System Modeling: Interactive Timing Analysis with Hotspot Highlighting". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS '16. Brest, France: Association for Computing Machinery, 2016, pp. 289–298. ISBN: 9781450347877. DOI: 10.1145/2997465.2997467. URL: https://doi.org/10.1145/2997465.2997467.

[75] Richard Fujimoto. *Parallel and Distributed Simulation Systems*. Hoboken, NJ, USA: John Wiley and Sons, 2000.

[76] Dan Gajski. *SpecC: Specification Language and Methodology*. Norwell, MA: Kluwer Academic Publishers, 2000.

[77] Abdoulaye Gamatie and Thierry Gautier. "The Signal Synchronous Multiclock Approach to the Design of Distributed Embedded Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 21.5 (2010), pp. 641–657.

[78] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. "Elements of reusable object-oriented software". In: *Reading: Addison-Wesley* (1995).

[79] Marc Geilen, Twan Basten, and Sander Stuijk. "Minimising Buffer Requirements of Synchronous Dataflow Graphs with Model Checking". In: *Design Automation Conference (DAC)*. ACM, 2005, pp. 819–824. DOI: 10.1145/1065579.1065796.

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

[81] David Gelernter. "Generative communication in Linda". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7.1 (1985), pp. 80–112.

[82] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. "Exploiting a natural network effect for scalable, fine-grained clock synchronization". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*. 2018, pp. 81–94.

[83] Michael Gibbs and Bjarne Stroustrup. "Fast dynamic casting". In: *Software: Practice and Experience* 36.2 (2006), pp. 139–156.

[84] Irene Greif. "Semantics of communicating parallel processes." PhD thesis. Massachusetts Institute of Technology, 1975.

[85] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999.

[86] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. "The Synchronous Data Flow Programming Language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1319.

[87] Philipp Haller and Martin Odersky. "Scala actors: Unifying thread-based and event-based programming". In: *Theoretical Computer Science* 410.2-3 (2009), pp. 202–220.

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

[89] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for Safety-critical Applications". In: *ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 372–383. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594310.

[90] David Harel. "Statecharts: A Visual Formalism for Complex Systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274.

[91] Ludovic Henrio, Einar Broch Johnsen, and Violet Ka I Pun. "Active Objects with Deterministic Behaviour". In: *International Conference on Integrated Formal Methods*. Springer. 2020, pp. 181–198.

[92] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. "Giotto: A Time-Triggered Language for Embedded Programming". In: *EMSOFT 2001*. Vol. LNCS 2211. Springer-Verlag, 2001, pp. 166–184.

[93] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. "Embedded Control Systems Development with Giotto". In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*. LCTES '01. Snow Bird, Utah, USA: Association for Computing Machinery, 2001, pp. 64–72. ISBN: 1581134258. DOI: 10.1145/384197.384208.

[94] Carl Hewitt. "Viewing control structures as patterns of passing messages". In: *Journal of Artificial Intelligence* 8.3 (1977), pp. 323–363.

[95] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*. 1973, pp. 235–245.

[96] C. A. R. Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: https://doi.org/10.1145/359576.359585.

[97] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84.

[98] "IEEE Standard VHDL Language Reference Manual". In: *ANSI/IEEE Std 1076-1993* (1994), pp. 1–288. DOI: 10.1109/IEEESTD.1994.121433.

[99] Shams M Imam and Vivek Sarkar. "Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries". In: *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*. 2014, pp. 67–80.

[100] IEEE Instrumentation and Measurement Society. *1588: IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. Report. IEEE, Nov. 2002.

[101] D. Jefferson. "Virtual Time". In: *ACM Trans. Programming Languages and Systems* 7.3 (1985), pp. 404–425.

[102] Chadlia Jerad and Edward A Lee. "Deterministic timing for the industrial internet of things". In: *2018 IEEE International Conference on Industrial Internet (ICII)*. IEEE. 2018, pp. 13–22.

[103] Gilles Kahn. "The Semantics of a Simple Language for Parallel Programming". In: *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974, pp. 471–475.

[104] Gilles Kahn and D. B. MacQueen. "Coroutines and Networks of Parallel Processes". In: *Information Processing*. Ed. by B. Gilchrist. North-Holland Publishing Co., 1977, pp. 993–998.

[105] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. "Autoware on board: Enabling autonomous vehicles with embedded systems". In: *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE. 2018, pp. 287–296.

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

[107] Hokeun Kim, Eunsuk Kang, David Broman, and Edward A. Lee. "Resilient Authentication and Authorization for the Internet of Things (IoT) Using Edge Computing". In: *ACM Trans. Internet Things* 1.1 (Mar. 2020). ISSN: 2691-1914. DOI: 10.1145/3375837. URL: https://doi.org/10.1145/3375837.

[108] Christoph M Kirsch and Ana Sokolova. "The logical execution time paradigm". In: *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.

[109] Hannes Klein. "Performance Evaluation of Reactor Programs". Bachelor's Thesis. TU Dresden, 2020.

[110] E. Kligerman and A. D. Stoyenko. "Real-Time Euclid: A language for reliable real-time systems". In: *IEEE Transactions on Software Engineering* SE-12.9 (Sept. 1986), pp. 941–949. ISSN: 2326-3881. DOI: 10.1109/TSE.1986.6313049.

[111] Philip Koopman. *A Case Study of Toyota Unintended Acceleration and Software Safety*. Blog. 2014. URL: http://betterembsw.blogspot.com/2014/09/a-case-study-of-toyota-unintended.html.

[112] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[113] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. "The time-triggered ethernet (TTE) design". In: *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. IEEE. 2005, pp. 22–33.

[114] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating Computer Simulation Systems: an Introduction to the High Level Architecture*. Prentice Hall PTR, 1999.

[115] Lindsey Kuper, Aaron Turon, Neelakantan R Krishnaswami, and Ryan R Newton. "Freeze after writing: Quasi-deterministic parallel programming with LVars". In: *ACM SIGPLAN Notices* 49.1 (2014), pp. 257–270.

[116] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. *Adversarial examples in the physical world*. 2017. arXiv: 1607.02533 [cs.CV].

[117] Leslie Lamport. "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems". In: *ACM Transactions on Programming Languages and Systems* 6.2 (1984), pp. 254–280.

[118] Leslie Lamport, Robert Shostak, and Marshall Pease. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[119] Elizabeth Latronico, Edward A. Lee, Marten Lohstroh, Chris Shaver, Armin Wasicek, and Matthew Weber. "A Vision of Swarmlets". In: *IEEE Internet Computing* 19.2 (2015), pp. 20–28. DOI: 10.1109/MIC.2015.17.

[120] E. A. Lee and D. G. Messerschmitt. "Synchronous Data Flow". In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245. DOI: 10.1109/PROC.1987.13876.

[121] E. A. Lee and T. M. Parks. "Dataflow Process Networks". In: *Proceedings of the IEEE* 83.5 (1995), pp. 773–801. DOI: 10.1109/5.381846.

[122] Edward Lee, Jan Reineke, and Michael Zimmer. "Abstract PRET machines". In: *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2017, pp. 1–11.

[123] Edward A Lee and Alberto Sangiovanni-Vincentelli. *The Tagged Signal Model - A Preliminary Version of a Denotational Framework for Comparing Models of Computation*. Tech. Report. EECS Department, University of California, 1996. URL: https://ptolemy.berkeley.edu/papers/96/denotational/.

[124] Edward A. Lee. "Computing Needs Time". In: *Communications of the ACM* 52.5 (2009), pp. 70–79. ISSN: UCB/EECS-2009-30. DOI: 10.1145/1506409.1506426.

[125] Edward A. Lee. "Cyber Physical Systems: Design Challenges". In: *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2008, pp. 363–369. DOI: 10.1109/ISORC.2008.25.

[126] Edward A. Lee. *EECS 219D: Semantics of Discrete-Event Systems*. https://bcourses.berkeley.edu/courses/1195544/files/folder/Lecture%20Notes?preview=45232443. Lecture Slides. Feb. 2014.

[127] Edward A. Lee. "Modeling Concurrent Real-time Processes Using Discrete Events". In: *Annals of Software Engineering* 7 (1999), pp. 25–45.

[128] Edward A. Lee. "The Problem with Threads". In: *Computer* 39.5 (2006), pp. 33–42. DOI: 10.1109/MC.2006.180.

[129] Edward A. Lee and Eleftherios Matsikoudis. "The Semantics of Dataflow with Firing". In: *From Semantics to Computer Science: Essays in memory of Gilles Kahn*. Ed. by Gérard Huet, Gordon Plotkin, Jean-Jacques Lévy, and Yves Bertot. Cambridge University Press, 2009.

[130] Edward A. Lee, Jan Reineke, and Michael Zimmer. "Abstract PRET Machines". In: *IEEE Real-Time Systems Symposium (RTSS)*. Dec. 2017.

[131] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Second. Cambridge, MA, USA: MIT Press, 2017. URL: http://LeeSeshia.org.

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

[133] Edward A. Lee and Haiyang Zheng. "Operational Semantics of Hybrid Systems". In: *Hybrid Systems: Computation and Control (HSCC)*. Ed. by Manfred Morari and Lothar Thiele. Vol. LNCS 3414. Springer-Verlag, 2005, pp. 25–53. DOI: 10.1007/978-3-540-31954-2_2.

[134] Edward Ashford Lee. *Plato and the Nerd — The Creative Partnership of Humans and Technology*. MIT Press, 2017.

[135] Stan Liao, Steve Tjiang, and Rajesh Gupta. "An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment". In: *Design Automation Conference*. ACM, Inc., 1997.

[136] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment". In: *Journal of the ACM* 20.1 (1973), pp. 46–61.

[137] Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance". In: *International Conference on Computer Design (ICCD)*. IEEE, 2012, pp. 87–93. URL: http://chess.eecs.berkeley.edu/pubs/919.html.

[138] J.W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000. ISBN: 9780130996510. URL: https://books.google.co.in/books?id=855QAAAAMAAJ.

[139] Xiaojun Liu and Edward A. Lee. "CPO semantics of timed interactive actor networks". In: *Theoretical Computer Science* 409.1 (2008), pp. 110–125. DOI: 10.1016/j.tcs.2008.08.044.

[140] Xiaojun Liu, Eleftherios Matsikoudis, and Edward A. Lee. "Modeling Timed Concurrent Systems". In: *CONCUR 2006 - Concurrency Theory*. Vol. LNCS 4137. Springer, 2006, pp. 1–15. DOI: 10.1007/11817949_1.

[141] M. Lohstroh, H. Kim, J. C. Eidson, C. Jerad, B. Osyk, and E. A. Lee. "On Enabling Technologies for the Internet of Important Things". In: *IEEE Access* 7 (2019), pp. 27244–27256. DOI: 10.1109/ACCESS.2019.2901509.

[142] M. Lohstroh and E. A. Lee. "Deterministic Actors". In: *2019 Forum for Specification and Design Languages (FDL)*. Sept. 2019, pp. 1–8. DOI: 10.1109/FDL.2019.8876922.

[143] M. Lohstroh, C. Menard, A. Schulz-Rosengarten, M. Weber, J. Castrillon, and E. A. Lee. "A Language for Deterministic Coordination Across Multiple Timelines". In: *2020 Forum for Specification and Design Languages (FDL)*. 2020, pp. 1–8. DOI: 10.1109/FDL50818.2020.9232939.

[144] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. "Reactors: A Deterministic Model for Composable Reactive Systems". In: *8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy'19)*. Vol. LNCS 11971. in press. Springer-Verlag, 2019.

[145] Marten Lohstroh and Edward A. Lee. "An Interface Theory for the Internet of Things". In: *International Conference on Software Engineering and Formal Methods (SEFM)*. Vol. LNCS 9276. Springer, 2015, pp. 20–34.

[146] Marten Lohstroh, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward 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, 152:1–152:4. ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3323469.

[147] Marten Lohstroh, Martin Schoeberl, Mathieu Jan, Edward Wang, and Edward A. Lee. "Work-in-Progress: Programs with Ironclad Timing Guarantees". In: *Proceedings of the International Conference on Embedded Software Companion*. EMSOFT '19. New York, New York: Association for Computing Machinery, 2019. ISBN: 9781450369244. DOI: 10.1145/3349568.3351553. URL: https://doi.org/10.1145/3349568.3351553.

[148] Carmen Torres Lopez, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. "Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper)". In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 27:1–27:30. ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.27. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10819.

[149] Matthew C. Loring, Mark Marron, and Daan Leijen. "Semantics of Asynchronous JavaScript". In: *SIGPLAN Not.* 52.11 (Oct. 2017), pp. 51–62. ISSN: 0362-1340. DOI: 10.1145/3170472.3133846. URL: https://doi.org/10.1145/3170472.3133846.

[150] Natividad Martínez Madrid, Peter T. Breuer, and Carlos Delgado Kloos. "A semantic model for VHDL-AMS". In: *Advances in Hardware Design and Verification: IFIP TC10 WG10.5 International Conference on Correct Hardware and Verification Methods, 16–18 October 1997, Montreal, Canada*. Ed. by Hon F. Li and David K. Probst. Boston, MA: Springer US, 1997, pp. 106–123. ISBN: 978-0-387-35190-2. DOI: 10.1007/978-0-387-35190-2_7. URL: https://doi.org/10.1007/978-0-387-35190-2_7.

[151] Oded Maler, Zohar Manna, and Amir Pnueli. "From Timed to Hybrid Systems". In: *Real-Time: Theory and Practice, REX Workshop*. Uses super dense time (super-dense, superdense). Springer-Verlag, 1992, pp. 447–484.

[152] Frédéric Mallet. "Clock constraint specification language: specifying clock constraints with UML/MARTE". In: *Innovations in Systems and Software Engineering* 4.3 (2008), pp. 309–314. DOI: https://doi.org/10.1007/s11334-008-0055-2.

[153] Louis Mandel, Cédric Pasteur, and Marc Pouzet. "ReactiveML, Ten Years Later". In: *Int. Symp. on Principles and Practice of Declarative Programming (PPDP)*. July 2015. DOI: 10.1145/2790449.2790509.

[154] Zohar Manna and Amir Pnueli. "Modeling Real Concurrency". In: *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992, pp. 103–175.

[155] Zohar Manna and Amir Pnueli. "Verifying Hybrid Systems". In: *Hybrid Systems*. Vol. LNCS 736. 1993, pp. 4–35.

[156] Eleftherios Matsikoudis and Edward A. Lee. "An Axiomatization of the Theory of Generalized Ultrametric Semilattices of Linear Signals". In: *International Symposium on Fundamentals of Computation Theory (FCT)*. Vol. LNCS 8070. Springer, 2013, pp. 248–258.

[157] Eleftherios Matsikoudis and Edward A. Lee. *The Fixed-Point Theory of Strictly Causal Functions*. Report UCB/EECS-2013-122. EECS Department, University of California, Berkeley, June 2013. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-122.html.

[158] Eleftherios Matsikoudis and Edward A. Lee. "The fixed-point theory of strictly causal functions". In: *Theoretical Computer Science* 574 (2015), pp. 39–77.

[159] Eleftherios Matsikoudis, Christos Stergiou, and Edward A. Lee. "On the Schedulability of Real-Time Discrete-Event Systems". In: *International Conference on Embedded Software (EMSOFT)*. ACM, 2013. DOI: 10.1109/EMSOFT.2013.6658590.

[160] Tobias R Mayer, Lionel Brunie, David Coquil, and Harald Kosch. "On reliability in publish/subscribe systems: a survey". In: *International Journal of Parallel, Emergent and Distributed Systems* 27.5 (2012), pp. 369–386.

[161] Erik Meijer. "Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues". In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, Maryland: ACM, 2010, 11:1–11:1. ISBN: 978-1-4503-0516-7.

[162] Christian Menard, Andrés Goens, Marten Lohstroh, and Jeronimo Castrillon. "Achieving Derterminism in Adaptive AUTOSAR". In: *Design, Automation and Test in Europe (DATE 20)*. in press. Grenoble, France, Mar. 2020.

[163] David L. Mills. *Computer Network Time Synchronization — The Network Time Protocol*. Boca Raton, FL: CRC Press, 2006.

[164] Robin Milner. "A calculus of communicating systems". In: (1980).

[165] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[166] Robin Milner. *Communication and concurrency*. Vol. 84. Prentice hall Englewood Cliffs, 1989.

[167] V. Milutinovic, J. Protic, and M. Tomasevic. "Distributed Shared Memory: Concepts and Systems". In: *IEEE Concurrency (out of print)* 4.02 (Apr. 1996), pp. 63–79. ISSN: 1558-0849. DOI: 10.1109/88.494605.

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

[169] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. "Effective static deadlock detection". In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 386–396.

[170] Walid A. Najjar, Edward A. Lee, and Guang R. Gao. "Advances in the dataflow computational model". In: *Parallel Computing* 25.13-14 (Dec. 1999), pp. 1907–1929. DOI: 10.1016/S0167-8191(99)00070-8.

[171] NASA Engineering and Safety Center. *National Highway Traffic Safety Administration Toyota Unintended Acceleration Investigation*. Technical Assessment Report. NASA, Jan. 2011.

[172] Saranya Natarajan and David Broman. "Timed C: An Extension to the C Programming Language for Real-Time Systems". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2018, pp. 227–239. DOI: 10.1109/RTAS.2018.00031.

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

[174] Marie-Agnes Peraldi-Frati and Julien DeAntoni. "Scheduling Multi Clock Real Time Systems: From Requirements to Implementation". In: *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 2011, pp. 50–57.

[175] Sibylla Priess-Crampe and Paulo Ribenboim. "Generalized Ultrametric Spaces I". In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 66 (1996), pp. 55–73.

[176] Aleksandar Prokopec. "Pluggable Scheduling for the Reactor Programming Model". In: *Programming with Actors: State-of-the-Art and Research Perspectives*. Ed. by Alessandro Ricci and Philipp Haller. Springer International Publishing, 2018, pp. 125–154. ISBN: 978-3-030-00302-9. DOI: 10.1007/978-3-030-00302-9_5.

[177] Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. "Flowpools: A lock-free deterministic concurrent dataflow abstraction". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2012, pp. 158–173.

[178] Aleksandar Prokopec and Martin 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. ISBN: 978-1-4503-3688-8. DOI: 10.1145/2814228.2814245.

[179] Claudius Ptolemaeus. *System Design, Modeling, and Simulation using Ptolemy II*. Berkeley, CA: Ptolemy.org, 2014. ISBN: 978-1-304-42106-7. URL: http://ptolemy.org/books/Systems.

[180] Peter Puschner and Alan Burns. "A review of worst-case execution-time analyses". In: *REAL TIME SYSTEMS-AVENEI NJ-* 18.2/3 (2000), pp. 115–128.

[181] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. "ROS: an open-source Robot Operating System". In: vol. 3. Jan. 2009.

[182] George M. Reed and A. W. Roscoe. "Metric Spaces as Models for Real-Time Concurrency". In: *3rd Workshop on Mathematical Foundations of Programming Language Semantics*. 1988, pp. 331–343.

[183] Shangping Ren and Gul A Agha. "RTsynchronizer: language support for real-time specifications in distributed systems". In: *ACM Sigplan Notices* 30.11 (1995), pp. 50–59.

[184] Leila Ribeiro Korff and Martin Korff. "True Concurrency = Interleaving Concurrency + Weak Conflict". In: *Electronic Notes in Theoretical Computer Science* 14 (1998). US-Brazil Joint Workshops on the Formal Foundations of Software Systems, pp. 204–213. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(05)80237-3. URL: http://www.sciencedirect.com/science/article/pii/S1571066105802373.

[185] Raymond Roestenburg, Rob Bakker, and Rob Williams. *Akka In Action*. Manning Publications Co., 2016.

[186] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just Model! – Putting Automatic Synthesis of Node-Link-Diagrams into Practice". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. San Jose, CA, USA, Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[187] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. "Patmos: A Time-predictable Microprocessor". In: *Real-Time Systems* 54(2) (Apr. 2018), pp. 389–423. ISSN: 1573-1383. DOI: 10.1007/s11241-018-9300-4.

[188] 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)*. Sept. 2018, pp. 5–16. DOI: 10.1109/FDL.2018.8524111.

[189] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Frédéric Mallet, Robert de Simone, and Julien Deantoni. "Time in SCCharts". In: *Proc. Forum on Specification and Design Languages (FDL '18)*. Munich, Germany, Sept. 2018.

[190] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. "Drawing Layered Graphs with Port Constraints". In: *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2 (2014), pp. 89–106. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2013.11.005.

[191] M. A. Sehr, M. Lohstroh, M. Weber, I. Ugalde, M. Witte, J. Neidig, S. Hoeme, M. Niknami, and E. A. Lee. "Programmable Logic Controllers in the Context of Industry 4.0". In: *IEEE Transactions on Industrial Informatics* (2020), pp. 1–1. DOI: 10.1109/TII.2020.3007764.

[192] Sanjit A Seshia and Jonathan Kotker. "GameTime: A toolkit for timing analysis of software". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2011, pp. 388–392.

[193] Sanjit A Seshia and Pramod Subramanyan. "UCLID5: Integrating modeling, verification, synthesis and learning". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2018, pp. 1–10.

[194] Sanjit A. Seshia and Alexander Rakhlin. "Quantitative Analysis of Systems Using Game-Theoretic Learning". In: *ACM Transactions on Embedded Computing Systems (TECS)* 11.S2 (2012), 55:1–55:27.

[195] Lui Sha, Abdullah Al-Nayeem, Mu Sun, José Meseguer, and Peter Ölveczky. *PALS: Physically Asynchronous Logically Synchronous Systems*. Report Technical Report. Univ. of Illinois at Urbana Champaign (UIUC), 2009.

[196] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020. ISBN: 9780078022159. URL: https://www.db-book.com/db7/index.html.

[197] Marjan Sirjani, Edward A Lee, and Ehsan Khamespanah. "Model checking software in cyberphysical systems". In: *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE. 2020, pp. 1017–1026.

[198] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. "Modeling and Verification of Reactive Systems using Rebeca". In: *Fundam. Inform.* 63.4 (2004), pp. 385–410.

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

[200] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[201] Alexander D. Stoyenko. "The evolution and state-of-the-art of real-time languages". In: *Journal of Systems and Software* 18.1 (1992), pp. 61–83. ISSN: 0164-1212. DOI: https://doi.org/10.1016/0164-1212(92)90046-M.

[202] System C Standardization Working Group and others. *1666-2011-IEEE Standard for Standard SystemC Language Reference Manual*.

[203] Samira Tasharofi, Peter Dinges, and Ralph E Johnson. "Why do scala developers mix the actor model with other concurrency models?" In: *European Conference on Object-Oriented Programming*. Springer. 2013, pp. 302–326.

[204] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. "TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs". In: *Formal Techniques for Distributed Systems*. Ed. by Holger Giese and Grigore Rosu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 219–234. ISBN: 978-3-642-30793-5.

[205] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S.V. 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.

[206] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[207] Stefan Tilkov and Steve Vinoski. "Node. js: Using JavaScript to build high-performance network programs". In: *IEEE Internet Computing* 14.6 (2010), pp. 80–83.

[208] TimeSys. *Real-Time Specification for Java, Reference Implementation*. Available at http://www.timesys.com/.

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

[210] A. M. Turing. "On Computable Numbers with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 42 (1936), pp. 230–265.

[211] John David Valois. "Lock-free data structures". In: (1996).

[212] Carlos Varela and Gul Agha. "Programming dynamically reconfigurable open systems with SALSA". In: *ACM SIGPLAN Notices* 36.12 (2001), pp. 20–34.

[213] Reinhard Von Hanxleden, Timothy Bourke, and Alain Girault. "Real-time ticks for synchronous programming". In: *2017 Forum on Specification and Design Languages (FDL)*. IEEE. 2017, pp. 1–8.

[214] Andrew Wellings. *Concurrent and real-time programming in Java*. John Wiley & Sons, Inc., 2004.

[215] Jonatan Wiik, Johan Ersfolk, and Marina Waldén. "A Contract-Based Approach to Scheduling and Verification of Dynamic Dataflow Networks". In: *2018 16th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2018, pp. 1–10.

[216] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. "The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools". In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), pp. 1–53. ISSN: 1539-9087. DOI: http://doi.acm.org/10.1145/1347375.1347389.

[217] R. Wittig, A. Goens, C. Menard, E. Matus, G. P. Fettweis, and J. Castrillon. "Modem Design in the Era of 5G and Beyond: The Need for a Formal Approach". In: *2020 27th International Conference on Telecommunications (ICT)*. 2020, pp. 1–5. DOI: 10.1109/ICT49546.2020.9239539.

[218] William Wulf and Mary Shaw. "Global variable considered harmful". In: *ACM Sigplan notices* 8.2 (1973), pp. 28–34.

[219] W. Xiang, P. C. Richardson, C. Zhao, and S. Mohammad. "Automobile Brake-by-Wire Control System Design and Analysis". In: *IEEE Transactions on Vehicular Technology* 57.1 (2008), pp. 138–145. DOI: 10.1109/TVT.2007.901895.

[220] R. K. Yates. "Networks of Real-Time Processes". In: *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*. Ed. by E. Best. Vol. LNCS 715. Springer-Verlag, 1993.

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

[222] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. 2nd. Discrete event systems (DEVS). Academic Press, 2000.

[223] Yang Zhao, Edward A. Lee, and Jie Liu. "A Programming Model for Time-Synchronized Distributed Real-Time Systems". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2007, pp. 259–268. DOI: 10.1109/RTAS.2007.5.

[224] Ye Zhou and Edward A. Lee. "Causality Interfaces for Actor Networks". In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), pp. 1–35. DOI: 10.1145/1347375.1347382.

[225] Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. "FlexPRET: A Processor Platform for Mixed-Criticality Systems". In: *Real-Time and Embedded Technology and Application Symposium (RTAS)*. 2014.

Appendix A

Summary of the Reactor Model

Execution environment	
Set of action instances	A
Set of identifiers	Σ (an abstract set)
Set of port instances	P
Set of priorities	$\mathbb{P} = \mathbb{Z}^- \cup \mathbb{Z}^+ \cup \{*\}$
Set of reaction instances	N
Set of reactor instances	R
Set of reactor classes	$\mathscr{C} \subseteq \Sigma$
Set of tags	$\mathbb{G} = \mathbb{T} \times \mathbb{N}$
Set of values	V (an abstract set)
Absent value	$\varepsilon \in V$
Current tag	$g = (t, m) \in \mathbb{G}$
First tag, last tag	$g_{\text{start}}, g_{\text{stop}} \in \mathbb{G}$
Current physical time	$T \in \mathbb{T}$
Reactor instantiation counter	$c_{\text{inst}} \in \mathbb{N}$
Reactor instantiation function	$\nu : \mathscr{C} \times \mathbb{N} \to R$
Event queue	$\mathcal{Q}_E$
Reaction queue	$\mathcal{Q}_R$
Defunct reactor stack	$\mathcal{S}_D$
Reactors	
Reactor instance	$r = (I, O, \mathcal{A}, S, \mathcal{N}, \mathcal{M}, \mathcal{R}, \mathcal{P}, \{\bullet, \diamond\}) \in R$
Set of input ports for r	$I(r) \subseteq \{p \in P \mid C(p) = r\}$
Set of output ports for r	$O(r) \subseteq \{p \in P \mid C(p) = r\}$
Set of actions for r	$\mathcal{A}(r) \subseteq \{a \in A \mid C(a) = r\}$
Set of state variables for r	$S(r) \subseteq \Sigma \times V$
Set of reactions contained in r	$\mathcal{N}(r) \subseteq \{n \in N \mid C(n) = r\}$
Set of mutations contained in r	$\mathcal{M}(r) \subseteq \mathcal{N}(r)$
Set of contained reactors of r	$\mathcal{R}(r) \subseteq \{r' \in R \mid C(r') = r\}$

Priority function	$\mathcal{P}(r) : \mathcal{N}(r) \to \mathbb{P}$
Startup trigger for r	$\bullet(r)$
Shutdown trigger for r	$\diamond(r)$
Reactor containing reactor r	$C(r) \subseteq R$
Ports	
Port instance	$p = (x, v) \in P$
Port identifier	$x \in \Sigma$
Port value	$v \in V$
Reactions with p as a source	$\mathcal{N}(p) = \left\{ n \in \left(\left(\bigcup_{r \in \mathcal{R}(C(p))} \mathcal{N}(r) \right) \cup \mathcal{N}(C(p)) \right) \;\middle|\; p \in D(n) \right\}$
Reactions with p as an effect	$\mathcal{N}^\vee(p) = \left\{ n \in \left(\left(\bigcup_{r \in \mathcal{R}(C(p))} \mathcal{N}(r) \right) \cup \mathcal{N}(C(p)) \right) \;\middle|\; p \in D^\vee(n) \right\}$
Reactor containing p	$C(p) \subseteq R$
Actions	
Action instance	$a = (x, v, \mathfrak{o}, d, s, \mathfrak{p}) \in A$
Action identifier	$x \in \Sigma$
Action value	$v \in V$
Action origin	$\mathfrak{o} \in \{\text{Logical}, \text{Physical}\}$
Minimum delay	$d \in \{t \in \mathbb{T} \mid t \geq 0\}$
Minimum spacing	$s \in \{t \in \mathbb{T} \mid t \geq 0\} \cup \bot$
Spacing violation policy	$\mathfrak{p} \in \{\text{Defer}, \text{Drop}, \text{Replace}\}$
Last scheduled event	$\mathcal{L}(a) \subseteq (\{a\} \times V \times \mathbb{G}) \cup \bot$
Reactor containing a	$C(a) \subseteq R$
Events	
Event instance	$e = (a, v, g)$
Event action	$a \in A$
Event value	$v \in V$
Event tag	$g \in \mathbb{G}$
Triggered reactions	$\mathcal{T}(a) = \{n \in \mathcal{N}(C(a)) \mid a \in \mathcal{T}(n)\}$
Reactions	
Reaction instance	$n = (D, \mathcal{T}, B, D^\vee, H, \Delta, B_\Delta) \in N$
Set of reaction sources	$D(n) \subseteq I(C(n)) \cup \left(\bigcup_{r \in \mathcal{R}(C(n))} O(r) \right)$
Set of reaction triggers	$\mathcal{T}(n) \subseteq D(n) \cup \mathcal{A}(C(n)) \cup \{\bullet, \diamond\}$
Reaction body	$B(n)$
Set of reaction effects	$D^\vee(n) \subseteq O(C(n)) \cup \left(\bigcup_{r \in \mathcal{R}(C(n))} I(r) \right)$
Set of schedulable actions	$H(n) \subseteq \mathcal{A}(C(n))$
Reactor containing reaction n	$C(n) \subseteq R$
Reaction priority	$\mathcal{P}(n) \in \begin{cases} \mathbb{Z}^- & \text{if } n \in \mathcal{M}(C(n)) \\ \mathbb{Z}^+ \cup \{*\} & \text{otherwise} \end{cases}$
Priority of unordered reactions	$\forall p \in \mathbb{Z}^- \, \forall q \in \mathbb{Z}^+ . (p < *) \wedge (q \not< *) \wedge (* \not< q) \wedge (* \leq *)$
Deadline	$\Delta(n) \in \{t \in \mathbb{T} \mid t \geq 0\} \cup \bot$
Deadline miss handler	$B_\Delta(n)$

Index