# Provable Determinism for Software in Cyber-Physical Systems

Marcus Rossel[1], Shaokai Jerry Lin[2], Marten Lohstroh[2], Jeronimo Castrillon[1], and Andrés Goens[34]

[1] TU Dresden, Germany {marcus.rossel,jeronimo.castrillon}@tu-dresden.de
[2] UC Berkeley, USA {shaokai,marten}@berkeley.edu
[3] The University of Edinburgh, UK
[4] University of Amsterdam, Netherlands a.goens@uva.nl

**Abstract.** In Cyber-Physical Systems (CPS), concurrently executing software components interact with each other and the physical environment to deliver functionality that is often safety-critical and time-sensitive. Verifying the correctness of the joint behavior of concurrent software components, however, is challenging. It is helpful to eliminate nondeterminism in the software, at the level of the programming model, and provide first-class programming constructs for expressing timed behavior. The Lingua Franca (LF) coordination language achieves this through the use of the Reactor model as its underlying model of computation. In this paper, we present the first formal operational semantics for the Reactor model, and prove its key properties of progress and determinism. The Reactor model and its associated proofs are fully mechanized in the Lean theorem prover. As an operational model, our semantics are close to the intuition for implementation and a helpful reference. The computational objects of the Reactor model are formalized in a modular fashion, which provides insights into the different structural properties of the model, and their effect on execution behavior.

## 1 Introduction

Cyber-physical systems (CPS) are systems where computational, digital components have integrated physical capabilities and interact with the physical world [4]. Designing and programming these systems brings about multiple challenges. The software has to make decisions about how to affect the physical environment, and the system has to deliver the intended behavior at the correct time. Describing the timed behavior of interacting digital components is one of the main challenges of modeling CPS [2,55,18], and are the subject of study of entire subfields, like timed automata [3] or the more general hybrid automata [22].

It is much easier to model CPS and design algorithms for them, if the behavior of the software can be understood as a function of the behavior in the physical environment. But for this to be true, the software has to respond deterministically to inputs from the physical environment. With the software in

CPS being increasingly concurrent, due to the use of multi-core, distributed, and networked system architectures, sophisticated coordination is required to ensure determinism in the software. Considering this kind of coordination as part of the application logic can lead to software that is brittle and complex, and tends to make the software less modular and more difficult to understand. Deterministic models of computation, which eliminate nondeterminism by construction, are an effective way to address this problem [30,31,19]. Such models have proven very useful in practice, particularly for the construction of CPS software [21,29,25].

Lingua Franca[5] (LF) [37] is a concrete open-source framework for programming CPS that has recently been gaining traction. The LF runtime enables deterministic concurrency without sacrificing performance [40], and the programming model has an explicit notion of time [32]. At the core of LF is a model of computation, the Reactor model [38,36,35], which offers deterministic timed semantics.

The Reactor model already has well-specified semantics, which ought to serve both as a specification for designing languages and be useful for reasoning about concrete programs. However, they are defined with complicated denotational semantics: using a superdense model of time [39], the semantics are defined through fixpoints of an ultrametric space [14]. In particular, the semantics are neither close to the implementation nor intuitive for programmers to understand. This is mitigated, in part, by the fact that parts of the specification are given as algorithms [36,35]. While the algorithms are useful for implementers, they lack the generality of a fully abstract operational semantics.

In this paper we consider an alternative formalization of the semantics of the Reactor model, using operational semantics [49]. Operational semantics are simpler to understand, while remaining generic – they don't tie the specification to a concrete implementation. For reasons like these, they have become widespread for defining the semantics of programming languages [45,46,57]. We define small-step operational semantics for the Reactor model, providing a foundation to the programming model that is simple and intuitive, yet rigorous.

With these operational semantics, we prove two key properties of the model: *progress* (execution does not get "stuck") and *determinism*. The model and its associated proofs are fully mechanized[6] in the Lean theorem prover [41]. This mechanization provides several advantages. First, it forces us to be precise about otherwise implicit definitions and assumptions, which is especially important in the context of verifying computation [28]. In particular, we find multiple ways in which the denotational semantics in [36,35] are imprecise, and make them precise. Second, it serves as a thorough documentation of the semantics, which provides insights into different structural properties of the model, and their consequences on execution behavior. For example, we require reactors to be finite to prove progress, while the proof of determinism does not need this restriction. Finally, it aids the process of working on modifications and extensions of the model, by uncovering where existing proofs break.

[5] https://www.lf-lang.org/
[6] https://github.com/lf-lang/reactor-model

The rest of this paper is structured as follows. Section 2 introduces the Reactor model by example and formalizes its computational objects. The operational semantics governing the execution model are given in Section 3, and its key properties are proven in Section 4. Finally, Section 5 covers related work and Section 6 concludes the paper.

## 2 Reactors

The Reactor model is a concurrent model of computation. Computation is encapsulated in *reactors*, the most fundamental structure of the model. To introduce reactors and their components informally, we walk through a model inspired by the Ingenuity Mars Helicopter[7] as shown in Figure 1. The helicopter uses multiple sensors, including different cameras which produce images that have to be processed. A controller then decides how to navigate based on these inputs and its internal logic, and controls actuators, like motors, to do so.

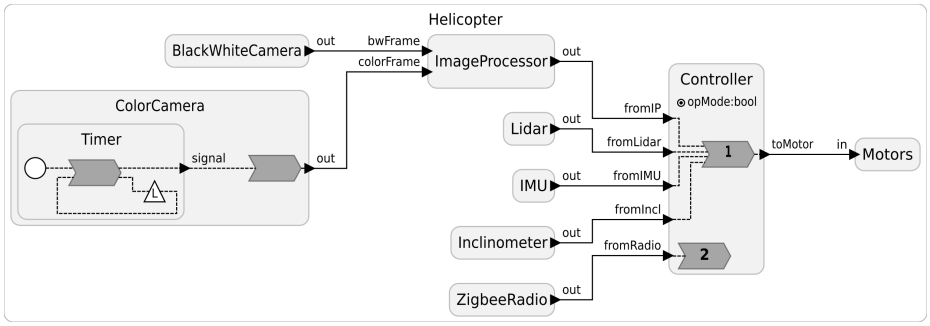


**Fig. 1.** Reactor-based model of the Ingenuity Mars Helicopter.

Each rounded rectangle in Figure 1 represents a reactor. We have expanded certain reactors like `ColorCamera` to show their components, while collapsing others like `ImageProcessor` where the details aren't relevant. The `ColorCamera` reactor models a camera which is continuously polled for new frames which are then communicated to the `ImageProcessor`. This communication is achieved by *connections* (depicted as bold lines) of *ports* (depicted as filled triangles). Concretely, the `ColorCamera` communicates frames via its `out` port, which has a connection to the `ImageProcessor`'s `colorFrame` port. Thus, when a value is set on `out` it is propagated to `colorFrame`. Ports are read from and written to by *reactions* (depicted as chevrons). Reactions are routines which can read and write values from and to different components called their *sources* and *effects*, which they must declare explicitly (depicted as dashed lines). When a source that is marked as a *trigger* has an available value, the reaction is *triggered*, executing the procedure which may read the values of its sources and set the

[7] See https://www.youtube.com/watch?v=D-Y6H0GMtbM&t=465s

values of its effects. In the example, the reaction contained in `ColorCamera` would be the routine responsible for actually reading the frame from the camera. It declares the `out` port as an effect, as it wants to write the frame to that port. It also declares the `signal` port of the *nested* `Timer` reactor as a source. That is, reactors can be nested in other reactors and thus form a tree structure. In fact, the entire model is represented by the single `Helicopter` reactor which contains all other reactors. The `Timer` reactor is used to implement polling by setting a value on its `signal` port every 33 milliseconds, which in turn triggers the `ColorCamera`'s reaction. To create an event delayed by 33 milliseconds, a *logical action* is used (depicted as a triangle containing the letter `L`). Whenever the `Timer`'s reaction executes, it *schedules* an event for the logical action with the given delay. After the specified amount of time, the logical action makes its value available which in turn triggers the reaction again. The circle shape contained in the `Timer` depicts a special kind of trigger that is present only at startup to bootstrap the process. Thus, upon starting the helicopter, the `Timer`'s output port periodically triggers the `ColorCamera`'s reaction which then supplies a steady stream of frames via its output port. These frames are propagated to and processed by the `ImageProcessor`, which finally communicates its results to the `Controller`. The `Controller` processes various inputs including the output of a `ZigbeeRadio`. This radio can be used to communicate a desired mode of operation to the `Controller`. For example, we may distinguish between a manual mode for testing on earth and an automatic mode for flight on Mars. The selected mode is stored locally in the `opMode` *state variable*. State variables can be accessed and modified only by a reactor's reactions. Thus, one of the `Controller`'s reactions is used to set the `opMode` based on the `ZigbeeRadio`'s input, while the other reads the `opMode` to determine which sensor values to include in its navigation algorithm.

The rest of this section will introduce reactors formally. As our formalization is mechanized in Lean, we express the following definitions in the language of Lean's underlying type theory [13,41], which is derived from the Calculus of Inductive Constructions (CIC) [16,17,42]. For simplicity of the presentation, we omit certain details and generally use more traditional mathematical (set-theoretic) syntax. However, the full details can be found in the accompanying mechanization, with the appendix linking to all presented definitions.

### 2.1 Basic Reactors

Formally, we define reactors in terms of axioms, similar to how algebraic structures like groups are defined in conventional mathematics. The canonical way to define objects in terms of axioms in Lean is by using type classes [54].[8] Thus, we define a hierarchy of type classes for reactors which successively add axioms until we arrive at what we call *proper* reactors. We split the axioms of reactors

[8] Type classes are similar to (but more powerful than) interfaces in Java, traits in Rust, protocols in Swift, etc.

into multiple type classes, as different parts of the formalization require varying degrees of constraints.

At the base of this type class hierarchy is the class of (basic) Reactors. It states that a reactor contains the following identifiable components: input ports, output ports, state variables, actions, reactions and nested reactors. By "identifiable" we mean that every component in a reactor has an associated identifier of some opaque ID type, which can be used to reference and obtain a component from a reactor. Formally:

**Definition 1 (Reactor).** A type $\alpha$ is a Reactor type if it has a partial function get?, which to any given element of $\alpha$, component kind $cpt$ and identifier associates an object of type $\mathsf{cptType}(cpt, \alpha)$. In Lean:

```
class Reactor (α : Type) where
  get? : α → (cpt : Component) → ID → Option (cptType cpt α)
```

Lean uses ML-syntax and currying, which means that "`cptType cpt α`" is the function application $\mathsf{cptType}(cpt, \alpha)$. The type $\mathsf{Option}(A)$ extends a type $A$ by the distinguished element none. Thus, we consider get? to be a partial function (a function only defined on a subset of its inputs) by returning none when the function is undefined for a given input. The Component type defines labels for the kinds of components listed above with the cptType function associating a type with each of these labels:

$$\mathsf{Component} = \{\mathsf{inp}, \mathsf{out}, \mathsf{stv}, \mathsf{act}, \mathsf{rcn}, \mathsf{rtr}\}$$

$$\mathsf{cptType}(cpt, \alpha) := \begin{cases} \alpha & \text{if } cpt = \mathsf{rtr} \\ \mathsf{Reaction} & \text{if } cpt = \mathsf{rcn} \\ \mathsf{Value} & \text{otherwise} \end{cases}$$

Thus, for example, $\mathsf{get?}(r, \mathsf{rcn}, i) = n$ means that reactor $r$ contains a reaction $n$ identified by identifier $i$. It is important to note that the notion of containment of a component in a reactor induced by get? is considered to be "flat". That is, components contained in nested reactors are not considered to be contained in the parent reactor. For example, Figure 2 shows that the reaction contained in the `Timer` reactor from Figure 1 is not considered to be contained in the `ColorCamera` reactor.[9]

[9] It *would* hold if `ColorCamera` also directly contained the reaction $n$ and identified it by $i$. In Section 2.2 we eliminate such duplicate identification with *hierarchical* reactors.

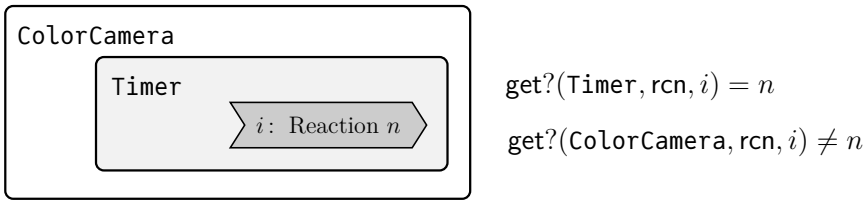


**Fig. 2.** Example of "flatness" of get?.

**Reactions** The component kinds inp, out, stv and act are similar by their associated type being Value. Thus, we collectively call them ValuedComponents. The Value type is an opaque type (in the same way as ID) of values with a distinguished absent element. Reactions, on the other hand, as the smallest units of computation in the Reactor model, are essentially functions with additional structure.

**Definition 2 (Reaction).** A *reaction* is a structure (tuple) of the form:[10]

```
structure Reaction where
  sources  : Set (ValuedComponent × ID)
  effects  : Set (ValuedComponent × ID)
  triggers : Set (ValuedComponent × ID)
  priority : Priority
  body     : (ValuedComponent → ID → Option Value) → List Change
```

We use the names of structures' fields to refer to the respective values. For example, given a reaction $n$ we write sources($n$) to denote the reaction's sources. The set of sources identifies components which are inputs to the reaction, and the set of effects are its outputs. A subset of the sources, called triggers, is used in the execution model to determine whether a reaction should be executed, with the priority affecting the order in which different reactions are executed.[11] At the heart of a reaction is its body, which is the function defining its behavior. Its input is a map providing the values of its sources. As an output, it returns *changes*, which formalize how a reaction affects its effects:

**Definition 3 (Change).** A Change combines an identified component with the value that should be assigned to it. Formally:[12]

```
structure Change where
  cpt : ValuedComponent
  id  : ID
  val : Value
```

[10] This definition omits certain edge cases for the sake of simplicity. For example, state variables are allowed as a sources, but not as a triggers.

[11] The Priority type is an opaque type with a partial order.

[12] This definition is slightly simplified. The type of the value actually depends on the specific valued component. For act the associated value has type Time × Value.

This can be seen as a special case of algebraic effects [48,26], as will become clearer in Section 3 when we define the execution semantics. We note that our rigorous definition of reaction bodies fills a gap in the formal treatment of reactions in [36,35], where they are defined simply as "executable code".

**Connections** Connections propagate values between reactors' ports. Notably, connections can only be established between nested reactors which live in the same parent reactor. We don't formalize connections directly, but instead take the same approach as [35] and replace them with the notion of a *relay reaction* as demonstrated in Figure 3. A relay reaction is a reaction whose sole purpose is to propagate a value from one port to another.

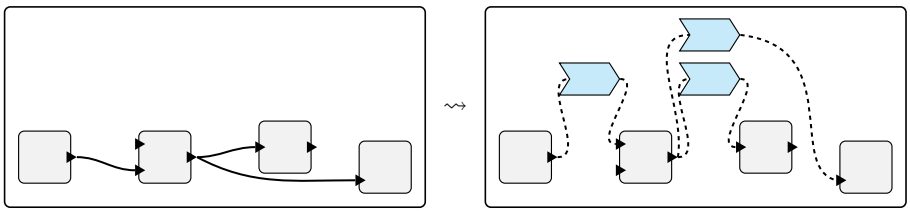

**Fig. 3.** Example of replacing connections by relay reactions.

**Definition 4 (Relay Reaction).** Let $r$ be a reactor with nested reactors $r_1$ and $r_2$ and $c$ be a connection from output port $o$ of $r_1$ to input port $i$ of $r_2$. A reaction $n$ is a *relay reaction* for $c$, if:

- $n$ is directly contained in $r$.
- $\mathsf{sources}(n) = \mathsf{triggers}(n) = \{(\mathsf{out}, o)\}$ and $\mathsf{effects}(n) = \{(\mathsf{inp}, i)\}$.
- $\mathsf{body}(n)$ writes the value of $(\mathsf{out}, o)$ to $(\mathsf{inp}, i)$.
- $\mathsf{priority}(n)$ is incomparable to the priorities of all other reactions in $r$.

The last requirement is crucial. As a connection propagates its values *immediately*, a corresponding relay reaction must be able to propagate its value at any time, as well. As a result, it must have a priority which allows it to execute independently of any other reaction in the same reactor.[13] In [35] this is achieved by defining a special priority which is only available to reactions for which it is known that they do not touch the reactor's state. This restriction is required as they define reactions in such a way that all state variables of their parent reactor are implicitly part of their sources and effects, and therefore all (normal) reactions in a reactor need to be totally ordered by their priority to retain determinism. In our formalization, this edge case is avoided by making

[13] This also implies that connections are only truly reducible to relay reactions if the Priority type has an incomparable element.

state variables *explicit* dependencies of reactions and defining rules for the ordering of reactions' priorities (Definition 8). Thus, relay reactions can be defined without requiring special considerations.

### 2.2 Hierarchical Reactors

Recall that nested reactors allow us to declare an entire system of reactors hierarchically, as a single root reactor that contains a tree of nested reactors. To reason formally about nested reactors and the resulting hierarchical structure, basic reactors are insufficient, as their get? function does not impose any structure on the components of reactors. For example, in basic reactors multiple components can share the same identifier, and the graph of reactors induced by nesting can form any directed graph.[14] As we intend to use identifiers to uniquely refer to components, we next constrain identifiers to be unique – this also forces the reactor graph to form a tree structure. We achieve this by forcing all components contained (arbitrarily deeply nested) in a reactor to be accessible by a unique path from the root reactor. A path from a reactor to an identified component is called a *membership witness*. Figure 4 shows how each step in a membership witness is a proof that we can get from one reactor to the next by direct nesting, ending in a proof that the desired identified component is contained in the final reactor.

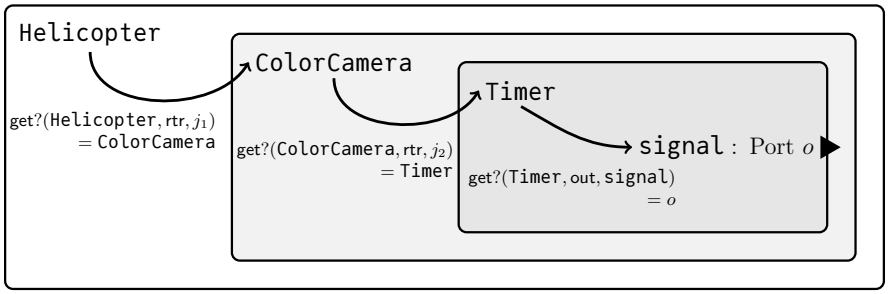


**Fig. 4.** Membership witness of the output port identified as `signal` for the `Helicopter` root reactor.

**Definition 5 (Member).** Formally, we define membership witnesses over a reactor type $\alpha$ inductively [17] as follows:

```
inductive Member (cpt : Component) (i : ID) : α → Type
  | final  : get? r cpt i = some o → Member cpt i r
  | nested : get? r₁ .rtr j = some r₂ → Member cpt i r₂ →
             Member cpt i r₁
```

In this Lean snippet, we assume that $\alpha$ is a basic reactor type, and we define a Member inductively as a function of a component kind *cpt*, an identifier

[14] For example, we can construct a self loop by letting $\mathsf{get?}(r, \mathsf{rtr}, i) = r$.

$i$ and a reactor of type $\alpha$. This inductive definition has two cases. The `final` (base) case has as condition that $\mathsf{get?}(r, cpt, i)$ is defined. The `nested` (inductive) case requires that the (root) reactor $r_1$ directly contains a nested reactor $r_2$ for which we already have a membership witness of $i$ of kind $cpt$. To construct a membership witness, we provide proofs of these conditions as arguments to the constructors, which are just functions by the principle of Propositions as Types [56,45]. By the same principle, a membership witness of $i$ of kind $cpt$ in $r$ is then a term of type $\mathsf{Member}(cpt, i, r)$.[15]

**Definition 6 (Hierarchical Reactors).** A Hierarchical reactor type is then a (basic) Reactor type where there exists at most one membership witness for any given identified component:

```
class Hierarchical (α) extends Reactor α where
  unique_ids : ∀ r cpt i (m₁ m₂ : Member cpt i r), m₁ = m₂
```

While hierarchical reactors impose structure on reactors, we yet need to define tools to aid formal reasoning over them. When defining properties over reactors, it is common to refer to components which are located *somewhere* in the tree of a given root reactor, but not necessarily in the same parent reactor. For example in Section 2.3 we define that "no two distinct reactions share an input port as effect." This requires us to be able to refer to any two reactions in the reactor tree. The get? function is ill-suited for this purpose, as its notion of containment is flat. That is, it only considers components *directly* nested in the reactor on which it is called.

Hierarchical reactors give us a natural way of extending the get? function such that we can refer to components anywhere in a reactor tree. For this, we first define the object of a membership witness to be the value of its identified component. For example if we have a membership witness $m : \mathsf{Member}(\mathsf{inp}, i, r)$ and the input port identified by $i$ has the value 5, then we define $\mathsf{object}(m) = 5$. We then define the partial function obj? which extends get? to work on an entire reactor tree:

**Definition 7 (Extended Accessor).**

$$\mathsf{obj?} : \alpha \to (cpt : \mathsf{Component}) \to \mathsf{ID} \to \mathsf{Option}(\mathsf{cptType}(cpt, \alpha))$$

$$\mathsf{obj?}(r, cpt, i) := \begin{cases} \mathsf{object}(m) & \text{if } m : \mathsf{Member}(cpt, i, r) \text{ exists} \\ \mathsf{none} & \text{if no membership witness exists} \end{cases}$$

Thus, for example, $\mathsf{obj?}(r, \mathsf{inp}, i) = 5$ means that somewhere in the tree of root reactor $r$ there exists an input port identified by $i$ with value 5. Note that this function is not computable in general, as we cannot decide the existence

[15] We note that Member lives in Type, not in Prop as we need to be able to distinguish different paths to the same identified component. If Member were a Prop, those paths would all be considered equal by proof irrelevance.

of a membership witness for an arbitrary hierarchical reactor type. For reactors containing only finitely many components it is easily computable, though.[16]

Hierarchical reactors are already vastly more useful than basic reactors in that they allow us to *define* many useful properties of reactors – often by using the extended accessor. For example, the entire execution model is defined over hierarchical reactors. To actually *prove* properties like progress and determinism, we need to add additional constraints provided in the following section. It is worth noting that formal treatment of identifier uniqueness, accessor functions and the distinction between properties needed for definitions as opposed to proofs are glossed over in the previous formal literature about the Reactor model [36,35].

### 2.3 Proper Reactors

The class of *proper* reactors adds all the axioms which are necessary to ensure that the execution of reactors can be defined in a way that results in deterministic behavior.

**Definition 8 (Proper Reactors).** A Proper reactor type is a hierarchical reactor type with the following additional properties:[17]

*(1) Unique Inputs* No two distinct reactions share an input port as effect.

*(2) Ordered Priorities* The priorities of any two distinct reactions in the same reactor are totally ordered if one of the following holds:

- The reactions share an effect.
- The reactions share a state variable as dependency which is an effect for at least one of them.

*(3) Valid Dependencies* Reactions only declare *valid* sources and effects. For a reaction $n$ contained directly in a reactor $r$, validity is defined as follows:

- Input ports of $r$ are valid sources for $n$.
- Output ports of $r$ are valid effects for $n$.
- State variables of $r$ are valid sources and effects for $n$.
- Actions and state variables of $r$ are valid sources and effects for $n$.
- Input ports of reactors directly nested in $r$ are valid effects for $n$.
- Output ports of reactors directly nested in $r$ are valid sources for $n$.

Notably, the ordering of priorities is not explicitly considered in [36,35] and the restrictions on state variables are new, as we now treat them as explicit dependencies.

[16] For example, by a simple tree traversal on the finite reactor tree.
[17] This is a simplification. See Section 2.4 for the full picture.

### 2.4 The Complete Type Class Hierarchy

The previous section omits some details of proper reactors. Namely, the class of proper reactors extends hierarchical reactors, as well as three others. These additional classes impose constraints which are not necessarily characteristic of reactors but are used for proving progress and determinism. In Figure 5 we show the full hierarchy of classes followed by brief explanations of each class.

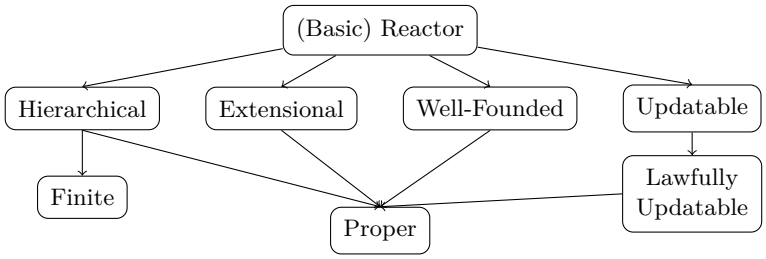


**Fig. 5.** Complete hierarchy of type classes used to formalize reactors.

*Extensional Reactors* A reactor type is called *extensional* if its reactors are completely characterized by their components:

$$\forall\, r_1, r_2 : (r_1 = r_2) \leftrightarrow (\forall\, cpt, i : \mathsf{get?}(r_1, cpt, i) = \mathsf{get?}(r_2, cpt, i))$$

*Well-Founded Reactors* A reactor type is called *well-founded* if the direct nesting relation $\exists\, i : \mathsf{get?}(r_2, \mathsf{rtr}, i) = r_1$ over $r_1$ and $r_2$ is well-founded [23,43]. That is, there are no infinitely deeply nested reactors.[18] This allows us to perform well-founded induction on reactors.

*(Lawfully) Updatable Reactors* A reactor type $\alpha$ is called *updatable* if it has a function $\mathsf{update} : \alpha \to \mathsf{ValuedComponent} \to \mathsf{ID} \to \mathsf{Value} \to \alpha$. It is called *lawfully updatable* if this function satisfies the intuitive notion of setting a given identified component to a given value.

*Finite Reactors* A hierarchical reactor type is called *finite* if each reactor contains only finitely many components. That is, the set $\{\, i \mid \exists\, o : \mathsf{obj?}(r, cpt, i) = o \,\}$ is finite for all $r$ and $cpt$.

We have thus defined reactors in a modular way, with the different properties shown in Figure 5. Overall, defining reactors this way serves two purposes. First, it formalizes implicit assumptions, like the uniqueness of identifiers or the proper ordering of reaction priorities. And second, it factors properties of reactors in a way that allows us to distinguish the necessary assumptions for main properties, like progress and determinism, as we will see in Section 4.

[18] A well-founded reactor can still contain infinitely many nested reactors by having an infinite branching factor.

## 3 Operational Semantics

Having defined reactors, we define their execution in this section by using operational semantics. We first consider the notions of time and dependencies, which are central to the Reactor model. We distinguish between *physical* (wall-clock) *time* and *logical time* [27,21,33]. Logical time is measured in terms of superdense-time *tags*, which consist of a *time value* and a *microstep* [39,34,5]. Execution of reactions (including writing to ports) and propagation of values via connections are considered "logically instantaneous", that is, they don't advance logical time. Thus, it is possible and typical for multiple reactions to execute at the same logical time. In this case, it is critical to determine an order for executing these reactions, which ensures that each reaction's dependencies have already finished executing before it itself executes. For this purpose, we define a *dependency relation* between reactions, based on their sources, effects and priorities:

**Definition 9 (Dependency Relation).** A reaction $n_1$ *is a dependency of* a reaction $n_2$ in reactor $r$, written $n_1 <_r n_2$, if one of the following holds:

1. $n_1$ and $n_2$ live in the same reactor and $\mathsf{priority}(n_1) > \mathsf{priority}(n_2)$.
2. There exist $i$ and a $cpt \neq \mathsf{stv}$, such that $(cpt, i) \in \mathsf{effects}(n_1) \cap \mathsf{sources}(n_2)$.[19]
3. There exists a reaction $n$ with $n_1 <_r n$ and $n <_r n_2$. That is, $<_r$ is transitive.

While reactions execute at a single instant in logical time, for the model to be useful, they also need to be able to communicate *across* logical time. Thus, we introduce *logical actions*, which allow reactions to schedule communication of values at specific logical time points in the future. Actions are similar to ports in that they carry values and can be read and written to by reactions, but differ from them in that they hold a given value at a specified future logical time. Thus, logical actions are the means by which future events are scheduled from *within* a reactor. Events can also be created asynchronously by the environment through *physical actions*. Such actions are not scheduled by reactions, but instead initiated by external components like sensors. Aside from how their events are created, physical actions and logical actions are handled equally in the model. The tags of events that originate from physical actions are *inputs* to the system. Hence, we do not need to model them as part of the semantics to conclude that the behavior triggered by such inputs is deterministic. For the sake of simplicity, we therefore do not include physical actions in our semantics.

Having understood the notions of time and dependencies, we can define execution as a relation that determines how to construct valid sequences of *execution states*. An execution state is a structure which adds context to a reactor for the purpose of managing its execution. More precisely:

**Definition 10 (Execution State).** An *execution state* over a hierarchical reactor type $\alpha$ is a structure of the following form:

[19] When $i$ is a state variable we get a dependency by combining Property 2 of proper reactors with Case 1 of $<_r$.

```
structure State (α) where
  rtr      : α
  tag      : Tag
  progress : Set ID
  events   : ID → Tag → Option Value
```

The rtr is the root reactor of the reactor system we want to execute. The tag holds the current logical time tag of the execution, where Tag is the type of logical time tags. The progress set indicates which reactions have already been processed at the current tag, while the events map keeps track of which action has which value at any given tag.

**Definition 11 (Execution Relation).** The operational semantics of reactor execution are then given by the *execution relation* $\downarrow^*$ over execution states. It is the reflexive, transitive closure of the execution step relation $\downarrow$ as shown in Figure 6.

$$\frac{}{s \downarrow^* s} \qquad \frac{s_1 \downarrow s_2 \quad s_2 \downarrow^* s_3}{s_1 \downarrow^* s_3} \qquad \frac{\mathsf{Allows}(s,n) \quad \neg\mathsf{Triggers}(s,n)}{s \downarrow \mathsf{record}(s,n)}\ \text{skip-step}$$

$$\frac{\mathsf{Allows}(s,n) \quad \mathsf{Triggers}(s,n) \quad s \text{ –}[\mathsf{output}(s,n)]\rightarrow s'}{s \downarrow \mathsf{record}(s',n)}\ \text{exec-step}$$

$$\frac{\mathsf{Closed}(s) \quad \mathsf{NextTag}(s,g) \quad \mathsf{Refresh}(\mathsf{rtr}(s),\, r,\, \mathsf{actions}(s,g))}{s \downarrow \langle r, g, \emptyset, \mathsf{events}(s)\rangle}\ \text{time-step}$$

**Fig. 6.** Operational semantics of reactor execution.

The *skip-step* and *exec-step* rules formalize how to process a single reaction. As they occur logically instantaneously, we collectively call them "instantaneous steps". In both cases, the reaction needs to be "allowed" to be processed, which is the case if all of $n$'s dependencies have been processed, but $n$ itself has not yet been processed. More formally:

**Definition 12 (Allows Relation).** An execution state $s$ *allows* a reaction $n$ to be processed, written $\mathsf{Allows}(s,n)$, if $\{\, n' \mid n' <_{\mathsf{rtr}(s)} n \,\} \subseteq \mathsf{progress}(s)$ and $n \notin \mathsf{progress}(s)$.

The choice of instantaneous step then depends on whether the reaction is currently triggered.

**Definition 13 (Triggering Relation).** A reaction $n$ is triggered at a given execution state $s$, written $\mathsf{Triggers}(s,n)$, if at least one of $n$'s triggers is present. That is, $\exists\,(cpt,i) \in \mathsf{triggers}(n) : \mathsf{obj?}(\mathsf{rtr}(s), cpt, i) \neq \mathsf{absent}$.

If a reaction $n$ is not triggered, *skip-step* applies, which records $n$ as being processed without executing it. That is, the execution state remains unchanged except for its progress which becomes $\mathsf{progress}(s) \cup \{n\}$. If on the other hand $n$ is triggered, *exec-step* applies, which executes the reaction and applies its resulting changes to the current execution state. The latter is formalized by the "update relation" $s \xrightarrow{[...]} s'$, which is satisfied if $s$ and $s'$ are equal up to applying a given list of changes. For the sake of brevity, we omit the details of this relation, but note that its definition is complicated by the Frame Problem [11]. That is, most of the difficulty in defining this relation arises from having to define what does *not* change when applying a change.

The given definitions of *skip-step* and *exec-step* give rise to nondeterministic choice in the semantics: at a given execution state, there may be multiple reactions which satisfy the conditions of either of these rules. The choice of which reaction to process next is then nondeterministic, which models the concurrency in the system. The fact that this degree of nondeterminism does not affect the resulting execution state at each time step is a key result of our semantics (see Section 4.2). Once all reactions have been processed at the current time tag, we call an execution state *closed*. A *time-step* can then occur, if there exists some future tag at which an event is scheduled:

**Definition 14 (Next-Tag Relation).** A tag $g$ is the *next tag* of an execution state $s$, written $\mathsf{NextTag}(s, g)$, if it is the smallest tag satisfying $\mathsf{tag}(s) < g$ and $\exists\, i : \mathsf{events}(s)(i, g) \neq \mathsf{none}$.

Performing a *time-step* on $s$ consists of advancing $\mathsf{tag}(s)$ to the next tag, setting $\mathsf{progress}(s) := \emptyset$, clearing the values of all ports and setting the values of all actions to the values given by $\mathsf{events}(s)$ for the new tag. The last two steps are handled by the Refresh relation.

**Definition 15 (Refresh Relation).** A hierarchical reactor $r_1$ *refreshes to* $r_2$ with action values $v : \mathsf{ID} \to \mathsf{Option}(\mathsf{Value})$, written $\mathsf{Refresh}(r_1, r_2, v)$, if:

- Inputs are cleared: $\forall\, i : \mathsf{obj?}(r_1, \mathsf{inp}, i) \neq \mathsf{none} \to \mathsf{obj?}(r_2, \mathsf{inp}, i) = \mathsf{absent}$
- Outputs are cleared: $\forall\, i : \mathsf{obj?}(r_1, \mathsf{out}, i) \neq \mathsf{none} \to \mathsf{obj?}(r_2, \mathsf{out}, i) = \mathsf{absent}$
- Action values are set: $\mathsf{obj?}(r_2, \mathsf{act}) = v$
- State variables are preserved: $\mathsf{obj?}(r_2, \mathsf{stv}) = \mathsf{obj?}(r_1, \mathsf{stv})$
- $r_1$ and $r_2$ are *structurally equivalent*. Intuitively, this means they contain the same components arranged equally and differ only by their values. The precise definition is rather technical and is therefore omitted here.[20]

## 4 Progress and Determinism

One of the major claims of the Reactor model is that, despite a certain degree of nondeterministic choice, its execution model is deterministic. In this section, we substantiate this claim and prove the following theorems over proper reactors:

[20] Cf. `Reactor.Equivalent` in the mechanization.

*Progress* There exists an execution step from any non-terminal execution state if and only if its reactor's dependency graph is acyclic.

*Determinism* All executions of a proper reactor up to the same point result in the same state. That is, the nondeterministic order of execution steps does not affect an execution's outcome.

We provide proof sketches for these theorems here; the full proofs can be found in our mechanization. The appendix lists the names of theorems and lemmas as they can be found in the mechanization.

### 4.1 Progress

In the literature surrounding the Reactor model, the definition of reactors usually includes the assumption that their dependency graphs are acyclic [36,35].[21] Yet, our formalization does not require this. In this section, we show that an acyclic dependency graph is only necessary to ensure that execution of a reactor can make progress. In fact, acyclicity of the dependency graph *characterizes* the ability to make progress. To prove this theorem, we first formalize its statement.

**Definition 16 (Progress Property).** We say that a reactor $r$ can make progress, that is, has the *progress property*, if for all non-terminal execution states $s$ starting at $r$ there exists a state $s'$ such that $s \downarrow s'$.

**Definition 17 (Terminal State).** An execution state $s$ is *terminal* if $\mathsf{progress}(s)$ contains all reactions of $\mathsf{rtr}(s)$ and there are no events scheduled after $\mathsf{tag}(s)$.

**Definition 18 (Dependency Graph).** The *dependency graph* of a reactor $r$ is the directed graph of reactions of $r$, where there is an edge between reactions $n_1$ and $n_2$ if and only if $n_1 <_r n_2$. As the dependency relation is transitive, the dependency graph of $r$ is acyclic if and only if the relation $<_r$ is irreflexive.

**Theorem 1 (Progress Theorem).** For all finite proper reactors $r$, $r$ has the progress property if and only if it has an acyclic dependency graph.

*Proof Sketch.* We prove both directions independently.

To prove the forward direction we need to show that for any reaction $n$, $n$ does not depend on itself ($n \not<_r n$). We achieve this by constructing a special non-terminal state $s$ over $r$ whose progress contains all reactions except $n$. Assuming the progress property for $r$, we obtain that there exists an execution step starting from $s$, which by construction of $s$ must be a *skip-step* or *exec-step* of $n$. In either case, $\mathsf{Allows}(s, n)$ must hold, from which one can easily prove that $n$ does not depend on itself.[22]

The backward direction is shown by explicitly constructing an execution step for a given non-terminal state $s$ over reactor $r$. If $s$ has no unprocessed reactions

[21] This is typically called an "absence of algebraic loops".

[22] The proof of this direction works generally for hierarchical reactors.

for the current tag, we can easily construct a *time-step*. Otherwise, we can show by induction on the set of unprocessed reactions that there must exist a reaction which is itself unprocessed, yet has only dependencies which are processed. This proof step works only because $r$'s dependency graph is acyclic. Depending on whether the obtained reaction is triggered or not, we can construct either a *skip-step* or *exec-step* for it.

### 4.2 Determinism

To formally state the theorem of determinism, we need to formalize the notion of "executions up to the same point". If we have executions $s \downarrow^* s_1$ and $s \downarrow^* s_2$, it is not sufficient to have only $\mathsf{tag}(s_1) = \mathsf{tag}(s_2)$. We also need to ensure that $s_1$ and $s_2$ have executed the same reactions within their current tag, that is, have the same progress:

**Theorem 2 (Determinism).** For all execution states $s$, $s_1$, $s_2$ over proper reactors, we have:

$$s \downarrow^* s_1 \to s \downarrow^* s_2 \to \mathsf{tag}(s_1) = \mathsf{tag}(s_2) \to \mathsf{progress}(s_1) = \mathsf{progress}(s_2) \to s_1 = s_2$$

*Proof Sketch.* The proof of this theorem can be divided into two main parts. First, we show that sequences of execution steps must always be structured as alternations of two kinds of subsequences as shown in Figure 7: (1) a sequence of instantaneous steps covering all reactions and (2) a single time step.

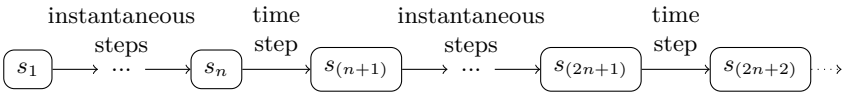


**Fig. 7.** Structure of executions as alternations of instantaneous steps and time steps.

This can be proven solely based on the structure of the execution rules and does not require the specific properties of proper reactors. Time steps can rather easily be proven deterministic, so what remains to be shown is that subsequences of instantaneous steps (denoted $\downarrow_{i+}$) are deterministic:

**Lemma 1 (Instantaneous Determinism).** For all execution states $s$, $s_1$, $s_2$ over proper reactors, we have:

$$s \downarrow_{i+} s_1 \to s \downarrow_{i+} s_2 \to \mathsf{progress}(s_1) = \mathsf{progress}(s_2) \to s_1 = s_2$$

This lemma lies at the heart of the proof of determinism. It is shown by induction using the following lemma, which establishes that if a reaction does not depend on another reaction, executing it first preserves the result of execution:

**Lemma 2 (Independent Reaction Swap).** For all instantaneous execution steps $e_1 : s_1 \downarrow_i s_2$ and $e_2 : s_2 \downarrow_i s_3$ with $\mathsf{rcn}(e_1) \not\leq_{\mathsf{rtr}(s_1)} \mathsf{rcn}(e_2)$, we can construct an execution state $s_2'$ and instantaneous execution steps $e_1' : s_1 \downarrow_i s_2'$ and $e_2' : s_2' \downarrow_i s_3$ such that $\mathsf{rcn}(e_1') = \mathsf{rcn}(e_2)$ and $\mathsf{rcn}(e_2') = \mathsf{rcn}(e_1)$.

We write $\mathsf{rcn}(e)$ to denote the reaction processed by instantaneous execution step $e$. As the statement of the lemma shows, we explicitly *construct* the swapped execution steps. As part of this construction we need to explicitly construct the new intermediate reactor $\mathsf{rtr}(s_2')$, which is why we need proper reactors to be lawfully updatable. Furthermore, the proof of this lemma builds on many small technical lemmas about independent reactions, which is where the properties of proper reactors are required. For example, they allow us to establish that independent reactions can never write to the same component.

**Corollary 1.** An immediate consequence of the theorem of determinism is that executions "synchronize" at every time step. That is, if we compare multiple executions we get a diamond structure as shown in Figure 8. It is an open problem to define execution in such a way that reordering of execution steps can transcend time barriers while retaining determinism.

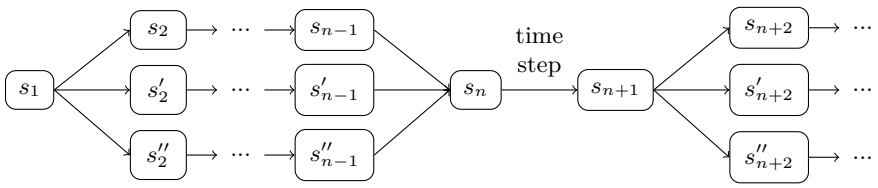


**Fig. 8.** Synchronization behavior of executions as a result of determinism.

## 5 Related Work

Concurrency is a central concept in computing in general, and consequently, there are many other well-studied models for describing and understanding concurrency. Petri nets are a staple formal model used for concurrency [44]. They are one of the most general models, specifying little in the way of structure. On the algebraic side, a family of models called Process Algebras [9] gives algebraic (equational) semantics to concurrent systems. These are particularly useful to build up the semantics compositionally, for example by introducing parallel or synchronous composition operators and specifying how the compound system behaves. Other models have also been put forward, like those based on game semantics [1] which all strive to the a high level of generality, allowing many behaviors that are undesirable in CPS and impossible in reactors. Kahn semantics [25], on the other hand, provide a topological view of deterministic concurrency based on Scott-continuous functions [53]. These models all enable modeling of concurrency, but not necessarily reasoning about time semantics.

Reactors mix concepts from several of these existing models, including actors [15], synchronous languages [50], dataflow [24], and discrete event systems [52]. They do so to enable deterministic, concurrent semantics with an explicit notion of time. Indeed, the Reactor model can also be characterized as a Sparse Synchronous Model [20] and a generalization of the Logical Execution Time (LET) paradigm [33], which has gained a lot of traction in industry.

Of particular interest here are synchronous languages, which are commonly used to program CPS. This includes Signal [7], Lustre [47] and commercial variants like SCADE or (a subset of) Simulink. These languages implement the synchronous/reactive model of computation, which can be seen as a subset of the discrete event semantics that are the basis of the Reactor model [51]. In general, synchronous languages like Lustre or Signal are more fine-grained than the Reactor model and have more restrictive semantics for the computational nodes. They also have a simpler model of time: discrete time, with an assumption of uniformity, as opposed to the superdense time model with an explicit distinction of logical and physical time in the Reactor model. Lustre, for example, forbids the use of unbounded loops and recursive functions. This is done to ensure a realistic synchrony with short response times. Signal similarly assumes an instantaneous execution of the digital logic in the system as well as the communication with the environment. This is in contrast to LF and the Reactor model, which models the execution times of reactions explicitly and allows arbitrary computation through potentially dynamic execution times.

The synchronous languages Lustre [8], Signal [7] and Esterel [10] were all defined with a specified semantics. Lustre's denotational semantics are in the Kahn style [25], which is the same style as the original reactor semantics [36], combined with an ultrametric space to model superdense time [14]. The denotational semantics of Signal are also based on Kahn Networks, but with extensions for their relational nature [6]. Interestingly, as in this paper, *both* Lustre *and* Signal also define operational semantics to be closer to the compiler implementation [47,7], even after having had denotational semantics before. Esterel was first given operational semantics in this style [10].

The operational semantics of Lustre are also mechanized in Coq for the Velus compiler [12]. A key difference is that these semantics (coinductively) reason about entire streams of execution at once, whereas our semantics reason about individual events in a stream. Moreover, Lustre and Signal are concrete languages, whereas the Reactor model is a more abstract model of computation; we formalize neither the concrete LF syntax nor its compiler.

## 6 Conclusion

LF and its underlying Reactor model enable constructing concurrent yet deterministic software for CPS. They provide primitives for concurrent processes, while enforcing structure through explicit dependencies and time semantics. We provide a rigorous formalization of the semantics in Lean and use it to deliver mechanized proofs of determinism and progress. Our proof of determinism shows

that the parallelism exposed by LF is safe, and our proof of progress provides a clear explanation as to why LF can only accept programs without causality loops. The operational semantics that we provide is simple and intuitive.

We envision that our work may prove useful for the construction of verified compilers and runtime implementations. This could enable verification of CPS software with a small trusted computing base (TCB) through Lean. Our formalization could also serve as tool for prototyping possible extensions of the core Reactor model and evaluating their consequences. An example of this would be developing a model of *mutations*, which are reactions that can change the structure of a reactor. The current mechanization already includes mutations as components, but ignores them in the execution model, as they require a significant reconsideration of the proof of determinism.

## Acknowledgments

We thank the anonymous reviewers for their feedback which greatly improved the clarity of the manuscript. This work was funded in part by the Engineering and Physical Sciences Research Council (EPSRC), through grant reference EP/V038699/1, as well as the German Federal Ministry of Education and Research (BMBF) as part of the Software Campus (01IS12051) and the program "Souverän. Digital. Vernetzt.", joint project 6G-life (16KISK001K). This work was also supported in part by the National Science Foundation (NSF), awards #CNS-1836601 (Reconciling Safety with the Internet) and #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems) and the iCyPhy Research Center (Indus- trial Cyber-Physical Systems), supported by Denso, Siemens, and Toyota.

## References

1. Abramsky, S., McCusker, G.: Game semantics. In: Computational Logic: Proceedings of the NATO Advanced Study Institute on Computational Logic, held in Marktoberdorf, Germany, July 29–August 10, 1997, pp. 1–55. Springer (1999)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical computer science **138**(1), 3–34 (1995)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical computer science **126**(2), 183–235 (1994)
4. Baheti, R., Gill, H.: Cyber-physical systems. The impact of control technology **12**(1), 161–166 (2011)
5. Bai, Y.: Desynchronization: From macro-step to micro-step. In: 2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE). pp. 1–10. IEEE (2018)
6. Benveniste, A., Le Guernic, P.: A denotational theory of synchronous communicating systems (1987)
7. Benveniste, A., Le Guernic, P., Jacquemot, C.: Synchronous programming with events and relations: the signal language and its semantics. Science of computer programming **16**(2), 103–149 (1991)

8. Bergerand, J.L.: LUSTRE: un langage déclaratif pour le temps réel. Ph.D. thesis, Institut National Polytechnique de Grenoble-INPG (1986)
9. Bergstra, J.A., Ponse, A., Smolka, S.A.: Handbook of process algebra. Elsevier (2001)
10. Berry, G., Cosserat, L.: The esterel programming language and its mathematical semantics. INRIA Res. Rep (327) (1984)
11. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Transactions on Software Engineering **21**(10), 785–798 (1995). https://doi.org/10.1109/32.469460
12. Bourke, T., Brun, L., Dagand, P.É., Leroy, X., Pouzet, M., Rieg, L.: A formally verified compiler for lustre. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 586–601 (2017)
13. Carneiro, M.: The type theory of lean (2019)
14. Cataldo, A., Lee, E., Liu, X., Matsikoudis, E., Zheng, H.: A constructive fixed-point theorem and the feedback semantics of timed systems. In: 2006 8th International Workshop on Discrete Event Systems. pp. 27–32. IEEE (2006)
15. Clinger, W.D.: Foundations of actor semantics. AITR-633 (1981)
16. Coquand, T., Huet, G.: The calculus of constructions. Information And Computation **76**(2-3) (1988)
17. Coquand, T., Paulin, C.: Inductively defined types. In: International Conference on Computer Logic. pp. 50–66. Springer (1988)
18. Cremona, F., Lohstroh, M., Broman, D., Lee, E.A., Masin, M., Tripakis, S.: Hybrid co-simulation: it's about time. Software & Systems Modeling **18**, 1655–1679 (2019)
19. Edwards, S.A.: On determinism. Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday pp. 240–253 (2018)
20. Edwards, S.A., Hui, J.: The sparse synchronous model. In: 2020 Forum for Specification and Design Languages (FDL). pp. 1–8. IEEE (2020)
21. Ernst, R., Kuntz, S., Quinton, S., Simons, M.: The logical execution time paradigm: New perspectives for multicore systems (dagstuhl seminar 18092). In: Dagstuhl Reports. vol. 8. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)
22. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings 11th Annual IEEE Symposium on Logic in Computer Science. pp. 278–292. IEEE (1996)
23. Hrbacek, K., Jech, T.: Introduction to set theory, revised and expanded. Crc Press (2017)
24. Jagannathan, R.: Dataflow models. Parallel and Distributed Computing Handbook pp. 223–238 (1995)
25. Kahn, G.: The semantics of a simple language for parallel programming. In: IFIP Congress (1974), https://api.semanticscholar.org/CorpusID:18030506
26. Kammar, O., Lindley, S., Oury, N.: Handlers in action. ACM SIGPLAN Notices **48**(9), 145–158 (2013)
27. Kirsch, C.M., Sokolova, A.: The logical execution time paradigm. Advances in Real-Time Systems pp. 103–120 (2012)
28. Lamport, L.: How to write a 21st century proof. Journal of Fixed Point Theory and Applications **11**(1), 43–63 (Mar 2012). https://doi.org/10.1007/s11784-012-0071-6, http://link.springer.com/10.1007/s11784-012-0071-6
29. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987). https://doi.org/10.1109/PROC.1987.13876
30. Lee, E.A.: The past, present and future of cyber-physical systems: A focus on models. Sensors **15**(3), 4837–4869 (2015)

31. Lee, E.A.: Determinism. ACM Transactions on Embedded Computing Systems (TECS) **20**(5), 1–34 (2021)
32. Lee, E.A., Lohstroh, M.: Time for all programs, not just real-time programs. In: Leveraging Applications of Formal Methods, Verification and Validation: 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17–29, 2021, Proceedings 10. pp. 213–232. Springer (2021)
33. Lee, E.A., Lohstroh, M.: Generalizing logical execution time. In: Principles of Systems Design. vol. LNCS 13660 (July 2023)
34. Lee, E.A., Zheng, H.: Operational semantics of hybrid systems. In: International Workshop on Hybrid Systems: Computation and Control. pp. 25–53. Springer (2005)
35. Lohstroh, M.: Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems (2020). https://doi.org/10.13140/RG.2.2.30520.78083, https://www.researchgate.net/publication/348155409
36. Lohstroh, M., Iñigó Romeo, I., Goens, A., Derler, P., Castrillon, J., Lee, E.A., Sangiovanni-Vincentelli, A.: Reactors: A Deterministic Model for Composable Reactive Systems (2020)
37. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a lingua franca for deterministic concurrent systems. ACM Transactions on Embedded Computing Systems (TECS) **20**(4), 1–27 (2021)
38. Lohstroh, M., Schoeberl, M., Goens, A., Wasicek, A., Gill, C., Sirjani, M., Lee, E.A.: Actors revisited for time-critical systems. In: Proceedings of the 56th Annual Design Automation Conference 2019. pp. 1–4 (2019)
39. Maler, O., Manna, Z., Pnueli, A.: Prom timed to hybrid systems. In: Real-Time: Theory in Practice: REX Workshop Mook, The Netherlands, June 3–7, 1991 Proceedings. pp. 447–484. Springer (1992)
40. Menard, C., Bateni, S., Donovan, P., Fournier, C., Lin, S., Suchert, F., Tanneberger, T., Kim, H., Castrillon, J., Lee, E.A.: High-performance deterministic concurrency using lingua franca. arXiv preprint arXiv:2301.02444 (2023)
41. de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28. pp. 625–635. Springer (2021)
42. Paulin-Mohring, C.: Introduction to the calculus of inductive constructions (2015)
43. Paulson, L.C.: Constructing recursion operators in intuitionistic type theory. Journal of Symbolic Computation **2**(4), 325–355 (1986)
44. Peterson, J.L.: Petri nets. ACM Computing Surveys (CSUR) **9**(3), 223–252 (1977)
45. Pierce, B.C.: Types and programming languages. MIT press (2002)
46. Pierce, B.C., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Software foundations. Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html (2010)
47. Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY. vol. 178, p. 188. Citeseer (1987)
48. Plotkin, G., Power, J.: Adequacy for algebraic effects. In: International Conference on Foundations of Software Science and Computation Structures. pp. 1–24. Springer (2001)
49. Plotkin, G.D.: A structural approach to operational semantics. Aarhus university (1981)

50. Potop-Butucaru, D., De Simone, R., Talpin, J.P.: The synchronous hypothesis and synchronous languages. The embedded systems handbook pp. 1–21 (2005)
51. Ptolemaeus, C.: System design, modeling, and simulation: using Ptolemy II, vol. 1. Ptolemy. org Berkeley (2014)
52. Ramadge, P., Wonham, W.: The control of discrete event systems. Proceedings of the IEEE **77**(1), 81–98 (1989). https://doi.org/10.1109/5.21072
53. Scott, D.S.: Domains for denotational semantics. In: Automata, Languages and Programming: Ninth Colloquium Aarhus, Denmark, July 12–16, 1982 9. pp. 577–610. Springer (1982)
54. Sozeau, M., Oury, N.: First-class type classes. In: Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings 21. pp. 278–293. Springer (2008)
55. Tabuada, P.: Verification and control of hybrid systems: a symbolic approach. Springer Science & Business Media (2009)
56. Wadler, P.: Propositions as types. Communications of the ACM **58**(12), 75–84 (2015)
57. Wadler, P.: Programming language foundations in Agda. In: Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21. pp. 56–73. Springer (2018)

## A Definitions and Theorems in the Lean Mechanization

Definition 1: `Objects/Reactor/Basic: Reactor`

Definition 2: `Objects/Reaction: Reaction`

Definition 3: `Objects/Change: Change.Normal`

Definition 4: not required in the mechanization

Definition 5: `Objects/Reactor/Basic: Reactor.StrictMember`

Definition 6: `Objects/Reactor/Hierarchical: Reactor.Hierarchical`

Definition 7: `Objects/Reactor/Hierarchical: Reactor.Hierarchical.obj?`

Definition 8: `Objects/Reactor/Proper: Reactor.Proper`

Definition 9: `Execution/Dependency: Dependency`

Definition 10: `Execution/State: Execution.State`

Definition 11: `Execution/Basic: Execution`

Definition 12: `Execution/State: Execution.State.Allows`

Definition 13: `Execution/State: Execution.State.Triggers`

Definition 14: `Execution/State: Execution.State.NextTag`

Definition 15: `Execution/Reactor: Reactor.Refresh`

Definition 16: `Execution/Theorems/Progress: Execution.Progress`

Definition 17: `Execution/State: Execution.State.Terminal`

Definition 18: `Execution/Dependency: Dependency`

Theorem 1: `Execution/Theorems/Progress: Execution.Progress.iff_deps_acyclic`

Theorem 2: `Execution/Theorems/Execution: Execution.deterministic`

Lemma 1: `Execution/Theorems/Grouped/Instantaneous: Execution.Instantaneous.Step.TC.deterministic`

Lemma 2: `Execution/Theorems/Grouped/Instantaneous: Execution.Instantaneous.Step.prepend_indep`