Polyglot Modal Models through Lingua Franca

Alexander Schulz-Rosengarten
Reinhard von Hanxleden
{als,rvh}@informatik.uni-kiel.de
Kiel University
Kiel, Germany

Marten Lohstroh
Edward A. Lee
{marten,eal}@berkeley.edu
UC Berkeley
Berkeley, USA

Soroush Bateni
soroush@utdallas.edu
UT Dallas
Dallas, USA

ABSTRACT

Complex software systems often feature distinct modes of operation, each designed to handle a particular scenario that may require the system to respond in a certain way. Breaking down system behavior into mutually exclusive modes and discrete transitions between modes is a commonly used strategy to reduce implementation complexity and promote code readability.

However, such capabilities often come in the form of self-contained domain specific languages or language-specific frameworks. The work in this paper aims to bring the advantages of modal models to mainstream programming languages, by following the polyglot coordination approach of Lingua Franca (LF), in which verbatim target code (e. g., C, C++, Python, Typescript, or Rust) is encapsulated in composable reactive components called reactors. Reactors can form a dataflow network, are triggered by timed as well as sporadic events, execute concurrently, and can be distributed across nodes on a network. With modal models in LF, we introduce a lean extension to the concept of reactors that enables the coordination of reactive tasks based on modes of operation.

CCS CONCEPTS

• **Software and its engineering** → *Model-driven software engineering*; *Abstraction, modeling and modularity*; *Visual languages*; **Orchestration languages**; **State based definitions**.

KEYWORDS

coodination, polyglot, modal models, state machines, Lingua Franca

ACM Reference Format:
Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Marten Lohstroh, Edward A. Lee, and Soroush Bateni. 2023. Polyglot Modal Models through Lingua Franca. In *Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week Workshops '23), May 09–12, 2023, San Antonio, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3576914.3587498

1 INTRODUCTION

The focus of this paper is on reactive systems, which continuously react to their environment, are typically embedded in larger systems, and often have some real-time requirements.

Two major notations or views have emerged for describing reactive systems, actor-oriented dataflow networks and state machines. The *dataflow view* breaks down the program into smaller blocks with streams of data flowing between them. Each such actor receives inputs, produces outputs, and can be assumed to operate fully independently from other blocks on which it has no data dependencies, thereby presenting opportunities for parallelization or distribution. MathWorks' Simulink and National Instruments' LabVIEW are examples for such an approach.

In a *state-oriented view*, the program is modeled in terms of states of the system and its progression in the form of transitions between them. State machine notations can be found, for example, as Stateflow [5] in Simulink or as Statecharts [6]. While state machines often describe fine-grained steps at the system level, they can also be used to represent more abstract *modes* of operation. For example, a system or subsystem may progress from initialization mode, through a training mode, and into a steady-state mode, with additional modes for error handling. Each mode of operation may encapsulate a complex collection of (stateful) reactive behaviors. Such *modal models* were realized, for example, in Ptolemy II [7], where they were used to simulate complex and hybrid systems.

However, the languages that provide the capabilities to model systems in any of these notations often come in the form of standalone domain specific languages (as in Simulink, LabVIEW, or Ptolemy II) or language-specific frameworks (such as Akka [15]). Usually, these languages either compile to or integrate into specific general purpose programming languages to produce executable code. The idea of *polyglot coordination* is to allow any mainstream programming languages to benefit from the advantages of modeling with actors, states, or modes. This can be done by directly embedding the verbatim code and then producing executable code that coordinates the execution of these modular units.

The goal of the work in this paper is to bring the advantages of modal models to mainstream programming languages through a reactor-oriented coordination language called Lingua Franca (LF) [12]. LF is rooted in a model of computation called *reactors* [11] and is built as a polyglot coordination language. Reactors encapsulate reactive tasks specified in verbatim code and provide a *minimal coordination layer* around them that is reactive, timed, concurrent, event-based, and accounts for isolated states. Unlike Ptolemy II, LF is not merely intended for modeling and simulation, but rather is meant for building efficient implementations. LF currently supports C, C++, Python, TypeScript, and Rust. For these languages it provides a runtime environment for automatic coordination of time-sensitive and concurrent or distributed reactors. The applicability of LF ranges from embedded systems to distributed systems deployed to the Cloud.

CPS-IoT Week Workshops '23, May 09–12, 2023, San Antonio, TX, USA

ACM ISBN 979-8-4007-0049-1/23/05…$15.00
https://doi.org/10.1145/3576914.3587498

Contributions and Outline

While the reactor-oriented modeling approach is well-suited for concurrent and distributed event-based coordination, it does not allow to naturally coordinate these tasks in terms of modes of operation, as we will illustrate in Sec. 2. After summarizing relevant aspects of LF in Sec. 3, we will present our concept of modal reactors, extending the polyglot coordination layer of LF. Specifically, this includes

- a lean textual and diagrammatic modal language extension that embraces the polyglot nature of LF by taking a "black-box approach" towards the target language and that allows hierarchical decomposition of modal behavior;
- an adaptation to reactors with two simple but effective transition types, reset and history, which can be further refined at the target language level; and
- a semantics for modal behavior that introduces mode-local time and leverages LF's superdense time model to achieve deterministic behavior.

Sec. 5 presents related work and Sec. 6 concludes.

We present the main aspects of modal reactors here. More detailed information, e. g. on the implementation, can be found in an extended report [17].

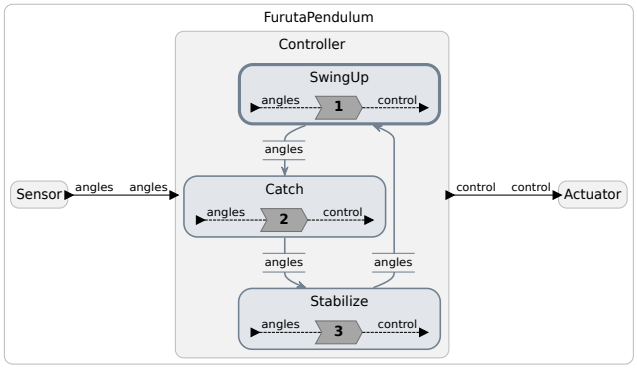
2 MOTIVATING EXAMPLE: THE FURUTA PENDULUM

A Furuta pendulum [4] is a classic control system problem often used to teach feedback control. The setup consists of a vertical shaft driven by motor, a fixed arm extending out at 90 degrees from the top of the shaft, and a pendulum at the end of the arm. The goal is to rotate the shaft to impart enough energy to the pendulum that it swings up, to then catch the pendulum and balance it so that the pendulum remains above the arm. Each of these steps requires a different control behavior which makes a controller a prime candidate for a modal model. It cycles through the three modes, which we will name `SwingUp`, `Catch`, and `Stabilize`.

From a classical event-driven or dataflow perspective, there is only a single reactive task, computing the motor control based on the angle measurements at the arm and shaft. However, with modes we can identify more fine-granular tasks and coordinate these by embedding them in a modal model.

To illustrate what we are aiming for with modal reactors, we have replicated a solution given by Eker et al. [9] and implemented it using our mode extension for Lingua Franca. The program is presented in Fig. 1. Our language extension includes the diagram synthesis capabilities of LF, which yields an automatically generated and interactive pictorial representation (Fig. 1a) of the textual program. The overall program consists of three connected reactors `Sensor`, `Controller`, and `Actuator`. Fig. 1b presents the source code of the top-level reactor. We will now explain the code for the `Controller` reactor, and, in the process, introduce Lingua Franca. Fig. 1c shows an abbreviated version of the `Controller` definition with some C code omitted for clarity. The source code of a more comprehensive implementation is available online.[1]

[1] https://github.com/lf-lang/examples-lingua-franca/tree/date23/C/src/modal_models/FurutaPendulum



(a) Structural overview as graphical diagram

```
target C;
import Controller from "FurutaPendulumController.lf"
import Sensor, Actuator from "FurutaPendulumUtil.lf"
main reactor {
    s = new Sensor();
    c = new Controller();
    a = new Actuator();
    s.angles -> c.angles;
    c.control -> a.control;
}
```

(b) The main reactor code for the program

```
target C;
reactor Controller {
  input angles:float[];
  output control:float;
  initial mode SwingUp {
    reaction(angles) -> control, reset(Catch) {=
      ... control law here in C ...
      lf_set(control, ... control value ... );
      if ( ... condition ... ) { lf_set_mode(Catch); }
    =}
  }
  mode Catch {
    reaction(angles) -> control, reset(Stabilize) {=
      ... control law here in C ...
      lf_set(control, ... control value ... );
      if ( ... condition ... ) { lf_set_mode(Stabilize); }
    =}
  }
  mode Stabilize {
    reaction(angles) -> control, reset(SwingUp) {=
      ... control law here in C ...
      lf_set(control, ... control value ... );
      if ( ... condition ... ) { lf_set_mode(SwingUp); }
    =}
  }
}
```

(c) The `Controller` reactor code

Figure 1: LF program to drive the Furuta Pendulum.

The very first line identifies the target language as C, which means that this controller will be translated into a standalone C module, and that the logic of reactions and mode transitions will be written in C. The first two lines in the `Controller` reactor define the input and output ports. Following are three *reaction* definitions, each reacting to the `angles` input, producing a control output, and implementing one of three control laws.

We here use the new mode extension to encapsulate each one in a separate mode. The reaction bodies are given in ordinary C code that reads the input values and calculates an actuation signal. That C code would go on lines 7, 14, and 21 but is abstracted away here because those details are not germane to this paper. On lines 8, 15, and 22, the calculated control value is sent to the output port. Lines 9, 16, and 23 use C expressions (abstracted here) to determine whether a mode change is now required, and, if so, invoke `lf_set_mode` to specify the next mode. See Sec. 4.2 for the semantics of these mode transitions.

With three modes, this model is a rather small example and could, as any statemachine, also be expressed in a single reaction that encodes the modal behavior (e. g. in a switch statement). Yet, such hand-written code would contradict the fundamental idea of model-driven engineering and would also be prone to errors and complex to extend. In more advanced modal scenarios, features like mode-local time and reset transitions (explained below) add expressiveness that is not easily replicated in non-modal LF.

3 REACTOR-ORIENTED PROGRAMMING

We now briefly introduce the structure of LF programs and highlight a few aspects that are especially relevant for the integration of modal models. For a more detailed discussion we refer elsewhere [10–12].

Causality and Concurrency. As the Furuta pendulum example already illustrates, *reactors* declare input and output *ports* and provide *reactions* to handle and produce events. Each reaction specifies *triggers* and *effects* while the actual body of a reaction is written in target code and not part of the LF coordination language layer. Reaction signatures are enforced and, hence, encode a causality interface [8]. In combination with reactor connections they form a dependency graph that captures all scheduling constraints necessary to ensure an execution that yields deterministic results. Because this graph is valid irrespective of the contents of the code that executes when reactions are triggered, reactions can be treated as a black box, which enables the polyglot nature of LF. Furthermore, the dependency graph allows to execute reactions in parallel without introducing any data races or deadlocks, if they are logically simultaneous and have no dependencies between them. Reactions within the same reactor always have dependencies between them, to ensure mutually exclusive access to shared state. Specifically, the order of declaration in the code determines precedence.

Time. Alongside ports and reactions, reactors can further contain *state variables*, *actions*, and *timers*. State variables provide access to stateful reactor-encapsulated data. Actions are used for scheduling of future events inside a reactor (as opposed to ports, which relay events logically instantaneously). Timers are actions that are automatically rescheduled with a predefined periodicity.

Events are aligned on a logical timeline, assigning each a *tag*. Tags are pairs (t, m), where t is a time value and m a microstep index. That index is used to enable subsequent event cycles to occur without any time elapsing between them, a concept known as *superdense time* [14]. Furthermore, LF features a special relationship between physical and logical time [13] to enable deadlines and handle external interrupts.

Visualization. The reactor-oriented programming paradigm lends itself particularly well to visualizations. LF comes with an automatic interactive diagram synthesis for its coordination layer, tailored to enhance developer's grasp of the high-level structure. One could say that LF capitalizes on "pragmatics" [19] when it comes to the handling of models of possibly highly complex systems, with a focus on how to get the best of both the textual and the graphical worlds. All the LF diagrams, including the ones presented in this paper, are synthesized automatically from textual LF code.

4 MODAL REACTORS

The basic idea of modal reactors is to use the existing reactor model but to allow for a modal coordination of reactions, by partitioning reactors into disjoint subsets that are associated with mutually exclusive *modes*. In a modal reactor, only a single mode can be active at a particular logical time instant, meaning that activity in other modes is automatically suspended. *Transitioning* between modes switches the reactor's behavior and controls the starting point of the entered modes. The option to reset or continue with the mode's history are common and powerful abstractions that are particularly helpful in managing complex timed behaviors, which can be extremely error-prone when carried out manually.

While the ideas behind modal reactors are not new, the guidance by LF's fundamental principles towards a polyglot modal coordination layer is novel. Our goal is to create modal models that are:

- **lean** a minimal coordination layer that provides the most essential functionality but still offers maximal versatility and user adjustability;
- **polyglot** a flexible multi-language wrapper that focuses on the user's language and requires only minor adaptation effort;
- **concurrent** allowing the design of multiple separate modal units acting independently;
- **timed** a reliable and precise way to specify time sensitive modal behavior, even in parallel and distributed environments; and
- **deterministic** yielding unambiguous and reproducible output behavior for the same sequence of tagged input events.

Our modal extension to LF embodies these very principles and embraces the crucial "black box" approach to reactions.

4.1 Syntax

The additional syntax required for adding modes is rather minimal. The added syntax allows reactions to be grouped into modes and includes new keywords for declaring (initial) modes and specifying transition types. The core LF language remains unchanged.

Modes can be defined in any reactor. Each mode requires a unique (per reactor) name and can declare contents that are local to this mode. There must be exactly one mode marked as `initial` (see line 5 in Fig. 1c). A mode can contain state variables, timers, actions, reactions, reactor instantiations, and connections. While the modes cannot be nested in other modes directly, hierarchical composition is possible through the instantiation of modal reactors. The main exception in allowed contents in modes are port declarations, as these are only possible on reactor level. Yet, modes share the scope with their reactor and, hence, can access ports, state variables, and

parameters of the reactor. Only the contents of other modes are excluded.

Mode transitions are declared within reactions. If a reactor has modes, reactions are allowed to list one or more as effects. This enables the use of the target language API to set the next mode, using `lf_set_mode` (e. g. in line 9 in Fig. 1c). The compiler will reject the program if the target code references a mode that is not declared as an effect. The user also has to specify the type of the transition by adding the modifier `reset` or `history` to the effect. An effect declared as `history(<mode>)` specifies a *history transition* to the mode, rendered in the graphical syntax with an "H" at the arrowhead (see Fig. 2a). The diagrams can also visualize the reaction triggers of transitions (e. g. angles in Fig. 1a), but due to the black-box nature of the target code in LF, the actual condition logic cannot be displayed with approach. The extended report [17] will elaborate on this design and further details of the syntax.

4.2 Modes and Transitions

The basic effect of modes in LF is that only parts that are contained in the currently active mode, or not contained in any mode, are executed at any point in time. This also holds for parts that are nested in multiple *ancestor modes* due to hierarchy; consequently, all those ancestors must be active in order to execute. Reactions in inactive modes are simply not executed. All components that model timing behavior, namely timers, scheduled actions, and delayed connections, are subject to a concept of *local time*. That means while a mode is inactive, the progress of time is suspended locally. How the timing components behave when a mode becomes active depends on the transition type. A mode can be *reset* upon entry, returning it to its initial state. Alternatively, if it was active before, it may continue based on its *history*. Sec. 4.3 will provide further insights to the concept of local time.

Upon reactor startup, the initial mode of each modal reactor is active, others are inactive. If at a tag (t, m), all reactions of this reactor and all its contents have finished executing, and a new mode was set in a reaction, the current mode will be deactivated and the new one will be activated for future execution. This means no reaction of the newly active mode will execute at tag (t, m); the earliest possible reaction in the new mode occurs one microstep later, at $(t, m + 1)$. Because of its superdense time model, LF is able to model a subsequent reaction at the same logical time, but one microstep later. Hence, if the newly active mode has for example a timer that will elapse with an offset of zero, it will trigger at $(t, m + 1)$. In case the mode itself does not require an immediate execution in the next microstep, it depends on future events, just as in the normal behavior of LF. Hence, modes in the same reactor are always mutually exclusive w. r. t. superdense time.

The introduction of a microstep delay gives order to subsequently activated modes and prevents causality issues that would occur if modes would be activated multiple times at the same tag. Yet, it still requires resolving potentially "conflicting" transition effects from different reactions. Here, we determine the effective target mode by relying on the fixed ordering of reactions within a reactor. In terms of deterministic outcome and overriding behavior, setting new modes can be considered analogous to assigning output ports. However, in terms of timing, transition effects correspond to scheduling actions with a zero delay, which also enforce a microstep delay to prevent causality cycles.

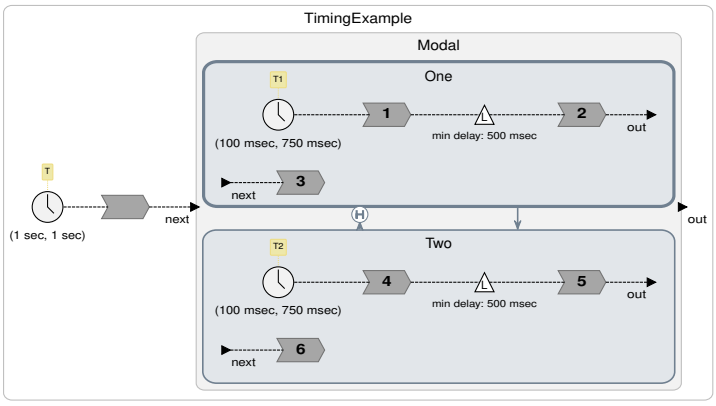
A transition is triggered if a new mode is set in a reaction body, as done on lines 9, 16, and 23 of Fig. 1c. The type of the transition is inferred from the effect declaration. In case a mode is entered with the reset behavior, all contained modal reactors are reset to their initial mode (recursively), all local timers are reset and start again awaiting their initial offset, all events (actions, timers, delayed connections) that were previously scheduled from within this mode are discarded, and a newly introduced `reset` trigger activates associated reactions in the mode and all contained reactors (recursively). Thus, whenever a mode is entered with a reset transition, the subsequent timing behavior is as if the mode was never executed before. State variables are subject to special handling via `reset` reactions, since it is idiomatic for reactors to use state variables to store manually managed resources. For history transitions, no reset is performed. This preserves its history and continues the behavior based on local time.

4.3 Local Time

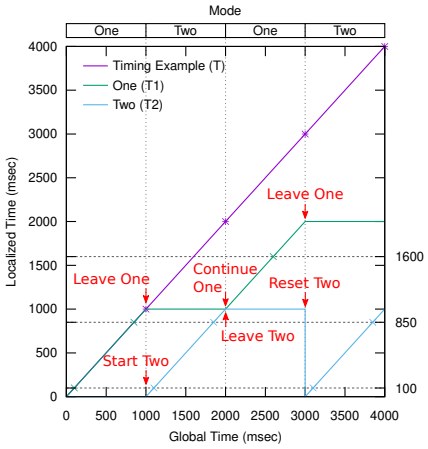
The notion of mode-local time, with the suspension of all timing behavior within inactive modes, is an established and well-formed principle also found in modal models in Ptolemy II [7]. The suspension of time gives a clear and consistent meaning to the inactivity of modes and provides a comprehensible state for the mode's contents upon entry. This especially favors modularity, as reactors that may be instantiated in modes do not have to anticipate the fact that their time (driven by timers or scheduled actions) will advance while their reactions are suppressed. Furthermore, modes allow to define reactor elements outside of modes, which gives the developer control over whether time should be local to a mode or not.

Fig. 2 illustrates the different characteristics of local time affecting timers and actions in the presence of the two transition types. Fig. 2a shows the generated diagram for a synthetic example program. It consists of two modes One (the initial mode) and Two, both in the Modal reactor. The next input toggles between these modes. It is driven by a reaction periodically triggered each second by timer T. The modes' contents are structured identically. Each has a timer T1/T2 that triggers a reaction after an initial offset of 100 msec and then periodically after 750 msec. This reaction then schedules a logical action with a delay of 500 msec. This action triggers the second reaction, which writes to the output out. The last reaction is triggered by the input next and invokes the transition to the other state. The main difference between the modes is that One is entered via a history transition, continuing its behavior, while Two is reset.

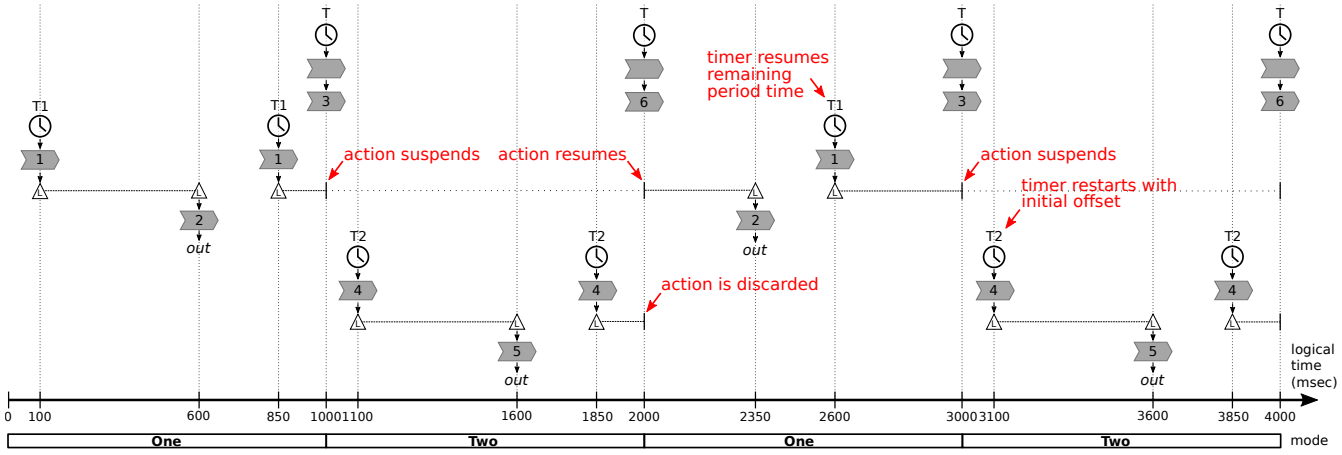
Fig. 2c illustrates the execution trace of the first 4 seconds of this program. Below the timeline is the currently active mode and above are the model elements that are executed at certain points in time, together with arrows indicating triggering relations and dashed lines for distribution through time. For example, at 100 msec, the initial offset of timer T1 elapses, which leads to the scheduling of the logical action in this mode. The action triggers the reaction 500 msec later, at 600 msec, and thus causes an output. The timing diagram illustrates the different handling of time between history transitions and reset transitions. Specifically, when mode One is



(a) The LF Model



(b) The progression of time in each mode and their respective timer



(c) The execution trace with reaction illustration

Figure 2: LF example illustrating the different effects of reset and history transitions on timers and delays in modes.

re-entered via a history transition, at time 2000 msec, the action triggered by T1 before, at time 850 msec, resumes. In contrast, when mode Two is re-entered via a reset transition, at time 3000 msec, the action triggered by T2 before, at time 1850 msec, gets discarded.

Fig. 2b illustrates the relation between global time in the environment and the localized time for each timer in Fig. 2. Since the top-level reactor TimingExample is not enclosed by any mode, its time always corresponds to the global time. Mode One is the initial mode and hence progresses in sync with TimingExample for the first second. During inactivity of mode One the timer is suspended and does not advance in time. At 2000 msec it continues relative to this time. T2 only starts advancing when the mode becomes active at 1000 msec. The reentry via reset at 3000 msec causes the local time to be reset to zero.

5 RELATED WORK

We are not aware of other work that aims to create a polyglot modal coordination layer based on a reactor-like foundation. However, there clearly is significant related work that our proposal builds on.

The synchronous modeling language SCADE [3] originally had a dataflow focus, where concurrently operating nodes communicate via streams. However, from version 6 onwards, it includes state machines as well, based on the mode extension proposed by Colaço et al. [2]. Their work is perhaps closest in spirit to what we propose here, also because they manage to enable modes through a minimal language extension. They extend boolean clocks, which control execution in Lustre/SCADE, into a richer type that encodes modes. They also employ a similar design to the mutual exclusion of modes, as described in Sec. 4.2. SCADE represents a standalone language and while it is able to integrate with its target languages and coordinate behavior through signals, it does not embody the rigorous polyglot coordination nature envisioned by LF.

More generally, there are many state machine notations, beyond flat finite-state machines (FSMs), that offer feature-rich language constructs. The most prominent ones are statecharts by Harel [6], which realize hierarchical state machines and are also part of UML. The statechart dialects that emerged in the context of synchronous languages, such as SyncCharts [1] or SCCharts [18], are of particular

relevance for modes in LF as their semantics also involve a clear notion of time and reactions. These languages provide a much broader set of transition types for immediate and delayed behavior and often include means to express preemptive behavior. While we have considered including such features, we decided to emphasize our lean approach. This especially, since we realized that many of these advanced aspects could be expressed in the target code by controlling transition logic and simulating their effects, which is not the case for the mechanics of reset and history transitions.

It is also common practice to express statecharts directly in classical programming languages without real language extensions. Samek describes how to express UML Statecharts in C/C++ [16]. As in UML Statecharts, this approach does not provide deterministic concurrency. Wagner et al. describe how to implement FSMs in C [20], but these are flat automata without any concurrency. Moreover, none of them follows a polyglot coordination approach.

Similarly, the Akka framework [15] provides means to write actor networks directly in Java, but without consideration of modes or determinism.

Sequential Function Charts, defined in the IEC 61131-3 standard for programming control logic, provide a state-transition model that can be combined with Function Block Diagrams. While this is similar to the way we can encapsulate modes in reactors, the standard does not aim for a polyglot approach.

6 CONCLUSION AND OUTLOOK

Modal reactors enable the coordination of reactive behavior in terms of modes and transitions. While we capitalize of many existing concepts of LF to achieve our objectives of being lean, polyglot, concurrent, timed, and deterministic, our design carefully adapts these fundamental principles and seamlessly integrates into the existing language, diagrams, and tooling. Modes are a fundamental concept in real-time systems and how designers think about them. They go beyond specifying low-level stateful behavior in state machines. Modal coordination in this context enables the orchestration of complex event processing networks associated with different modes of operation. Additionally, with mode-local time, it grants a powerful tool for modeling timed-behavior. Pausing and continuing mode-local behavior is a capability that is otherwise tedious to achieve in LF.

While modes are already central to a large family of existing programming and modeling languages, our approach of building modal abstractions into the polyglot coordination language LF has the advantage of being applicable to a range of target languages at once. Programmers can still develop the low-level "business logic" in any target language supported by LF, and may use LF solely to express modal aspects in a lean, deterministic manner, with diagramming support, largely irrespective of which other LF facilities might be harnessed as well. Our implementation currently provides modal support for the C and Python targets, demonstrating the versatility of our approach. While other targets will follow, the C target illustrates the suitability for embedded low-level applications, while Python shows compatibility with a high-level scripting language commonly used in Cloud and machine learning applications. We also successfully used modes in LF to control a robot and specify different modes for driving and collision avoidance.

The introduction of explicit modes in the model opens up new opportunities for static analyses. The LF compiler already considers mutual exclusion imposed by modes in the detection of causality issues. We plan to follow this avenue and develop tools for verification and model checking of modal reactors. Another direction is the exploration of modal models coordinating federated LF programs, as well as improving simulation and live visualization of modal LF.

REFERENCES

[1] Charles André. 2004. Computing SyncCharts Reactions. *Electronic Notes in Theoretical Computer Science* 88 (Oct. 2004), 3–19.

[2] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2006. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM International Conference on Embedded Software (EMSOFT'06)*. ACM, Seoul, South Korea, 73–82.

[3] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: A formal language for embedded critical software development (invited paper). In *11th International Symposium on Theoretical Aspects of Software Engineering TASE*. Sophia Antipolis, France, 1–11.

[4] Katsuhisa Furuta, M. Yamakita, and S. Kobayashi. 1992. Swing-up control of inverted pendulum using pseudo-state feedback. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 206, 4 (1992), 263–269.

[5] Grégoire Hamon and John Rushby. 2004. An operational semantics for Stateflow. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 229–243.

[6] David Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of computer programming* 8, 3 (1987), 231–274.

[7] Edward A. Lee and Stavros Tripakis. 2010. Modal Models in Ptolemy. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*, Vol. 47. Linköping University Electronic Press, Linköping University, 11–21.

[8] Edward A. Lee, Haiyang Zheng, and Ye Zhou. 2005. Causality interfaces and compositional causality analysis. *Foundations of Interface Technologies (FIT), Satellite to CONCUR, San Francisco, CA* 2 (2005), 402–405.

[9] Jie Liu, Johan Eker, Jörn W Janneck, and Edward A Lee. 2002. Realistic simulations of embedded control systems. *IFAC Proceedings Volumes* 35, 1 (2002), 391–396.

[10] Marten Lohstroh. 2020. *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems.* Ph. D. Dissertation. EECS Department, University of California, Berkeley.

[11] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. 2019. 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. Springer-Verlag, 27.

[12] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Trans. Embed. Comput. Syst.* 20, 4, Article 36 (May 2021), 27 pages.

[13] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A Lee. 2020. A language for deterministic coordination across multiple timelines. In *2020 Forum for Specification and Design Languages (FDL)*. IEEE, 1–8.

[14] Zohar Manna and Amir Pnueli. 1992. Verifying hybrid systems. In *Hybrid Systems*. Springer, 4–35.

[15] Raymond Roestenburg, Rob Bakker, and Rob Williams. 2016. *Akka In Action*. Manning Publications Co.

[16] Miro Samek. 2008. *Practical UML Statecharts in C/C++Event-Driven Programming for Embedded Systems.* Newnes.

[17] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Marten Lohstroh, Soroush Bateni, and Edward A. Lee. 2023. Modal Reactors. https://doi.org/10.48550/ARXIV.2301.09597 arXiv:2301.09597

[18] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. 2014. SCCharts: Sequentially Constructive Statecharts for Safety-critical Applications. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 372–383.

[19] Reinhard von Hanxleden, Edward A. Lee, Hauke Fuhrmann, Alexander Schulz-Rosengarten, Sören Domrös, Marten Lohstroh, Soroush Bateni, and Christian Menard. 2022. Pragmatics twelve years later: a report on Lingua Franca. In *11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA) (Lecture Notes in Computer Science, Vol. 13702)*. Springer, Rhodes, Greece, 60–89.

[20] Ferdinand Wagner, Ruedi Schmuki, Peter Wolstenholme, and Thomas Wagner Thomas. 2006. *Modeling Software with Finite State Machines: A Practical Approach*. Auerbach Publications.