模态反应器

Alexander Schulz-Rosengarten*、Reinhard von Hanxleden*、Marten Lohstroh†、Soroush Bateni‡、Edward A. Lee†
* 基尔大学，基尔，德国， {als, rvh}@informatik.uni-kiel.de
† 加州大学伯克利分校，加利福尼亚州，美国， {marten, eal}@berkeley.edu
‡ 达拉斯德克萨斯大学，得克萨斯州，美国，s
oroush@utdallas.edu

arXiv:2301.09597v1 [cs.PL] 23 Jan 2023

摘要——**复杂软件系统通常具有不同的运行模式，每种模式旨在处理特定场景，而该场景可能要求系统以某种方式作出响应。将系统行为划分为互斥的模式以及模式之间的离散转换，是降低实现复杂度并提高代码可读性的常用策略。**

然而，这类功能通常以独立的领域专用语言或特定语言的框架形式提供。本文旨在通过遵循 Lingua Franca（LF）的多语言协调方法，将模态模型的优势带入主流编程语言。该方法将原样的目标代码（例如 C语言、C++、Python、TypeScript 或 Rust）封装在称为反应器的可组合反应式组件中。反应器可以构成数据流网络，由定时事件和偶发事件触发，并发执行，还可以分布在网络中的各个节点上。

借助 LF 中的模态模型，我们对反应器这一概念进行了精简扩展，使其能够基于运行模式协调响应式任务。本文所概述的模态反应器实现方案，只需对通用运行时系统稍作修改，即可推广至 LF 支持的任何语言。

索引词—**协调、多语言、模态模型、状态机、模型驱动工程、反应器、Lingua Franca**

I. 引言

本文聚焦于反应式系统。这类系统会持续响应其环境，通常嵌入在更大的系统中，并且往往有一定的实时要求。

描述反应式系统的主要表示法或视图有两种：面向参与者的数据流网络和状态机。数据流视图 将程序分解为较小的模块，数据流在这些模块之间流动。每个参与者都会接收输入、生成输出，并且可以假定它完全独立于其他与其不存在数据依赖关系的模块运行，因此也为并行化或分布式部署创造了机会。MathWorks 的 Simulink 和 National Instruments 的 LabVIEW 就是这种方法的例子。

在面向状态的视角中，程序根据系统的状态及其状态转换过程进行建模。状态机表示法的例子包括 Simulink 中的 Stateflow [11]以及 Statecharts [12]。状态机通常用于描述系统层面的细粒度步骤，但也可以用来表示更抽象的模式 。例如，系统或子系统可以从初始化模式开始，经过训练模式，进入稳态模式，并设有用于错误处理的其他模式。每种运行模式都可以封装一组复杂的（有状态的）反应式行为。这类模态模型例如，这些模型在 Ptolemy II [17],中得以实现，并用于模拟复杂系统和混合系统。

然而，能够以这些表示法中的任意一种对系统建模的语言，往往是独立的领域专用语言（如 Simulink、LabVIEW 或 Ptolemy II），或特定语言框架（如 Akka [26]）。通常，这些语言要么编译为特定的通用编程语言，要么与之集成，以生成可执行代码。多语言协调旨在让任何主流编程语言都能受益于使用参与者、状态或模式进行建模的优势。为此，可以直接嵌入原样代码，然后生成能够协调这些模块化单元执行的可执行代码。

本文工作的目标是通过一种名为 Lingua Franca (LF) [22]、以反应器为核心的协调语言，将模态模型的优势带入主流编程语言。LF 源自一种称为反应器 [21] 的计算模型，并以多语言协调语言的形式构建。反应器封装以原样代码指定的反应式任务，并在其周围提供精简的协调层 ，使其具备反应式、定时、并发、基于事件的特性，并能够处理隔离状态。与 Ptolemy II 不同，LF 不只是用于建模和仿真，而是旨在构建高效实现。LF 目前支持 C、C++、Python、TypeScript 和 Rust。对于这些语言，LF 提供了一个运行时环境，用于自动协调时间敏感的并发或分布式反应器。LF 的适用范围从嵌入式系统到部署在云上的分布式系统。

贡献与概要

尽管以反应器为核心的建模方法非常适合并发和分布式的基于事件的协调，但它无法自然地按照运行模式来协调这些任务，正如我们将在第二节所说明的那样。在第三节总结 LF 的相关方面后，我们将介绍模态反应器的概念，以扩展 LF 的多语言协调层。具体而言，这包括

- 一种简洁的文本与图示模态语言扩展，它通过对目标语言采用“黑箱方法”来体现 LF 的多语言特性，并支持对模态行为进行分层分解；
- 对反应器的一种改造，加入了两种简单而有效的转换类型：重置和历史；它们还可以在目标语言层面进一步细化；以及

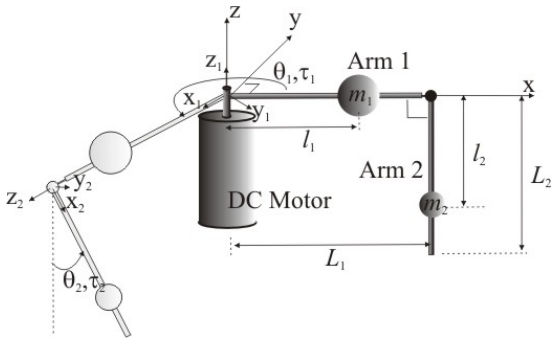


图 1. Benjamin Cazzolato 根据维基百科绘制的 Furuta 摆示意图 — 知识共享署名 3.0。

- 一种模态行为语义，它引入模式局部时间，并利用 LF 的超密集时间模型来实现确定性行为。

第 V 节更深入地探讨实现方面的问题，第 VI 节则详细阐述关键设计决策和潜在替代方案。我们在第 VII 节讨论相关工作，并在第 VIII 节总结。

II. 动机示例：Furuta 摆

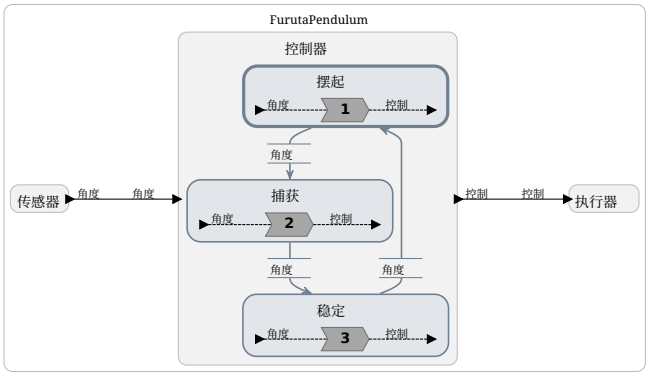
Furuta 摆 [10] 是一种经典的控制系统问题，常用于教授反馈控制。如第 II 节所示，它由一根电机驱动的垂直轴、一根从轴顶端以 90 度角向外延伸的固定臂，以及位于臂末端的摆锤组成。目标是转动轴，为摆锤注入足够的能量，使其摆起；随后捕获摆锤并使其保持平衡，让摆锤停留在臂的上方。这些步骤各自需要不同的控制行为，因此控制器非常适合采用模态模型。它会在三种模式之间循环，我们将其命名为 摆起、捕获和 稳定。

从经典的事件驱动或数据流视角来看，只有一个响应式任务：根据摆臂和轴的角度测量来计算电机控制。不过，借助模式，我们可以识别出更细粒度的任务，并将它们嵌入模态模型中加以协调。

为了展示我们对模态反应器的设想，我们复现了 Eker 等人提出的一种解决方案 [19]，并使用 Lingua Franca 的模式扩展实现了该方案。程序如图 2 所示。我们的语言扩展包含 LF 的图示合成功能，可自动生成文本程序的交互式图形表示（图2a）。整个程序由三个相互连接的反应器组成：传感器、控制器和执行器。下面我们将讲解该程序的代码，并借此介绍 Lingua Franca。图 2b 和图 2c 展示了一个经过简化的版本，为便于说明，其中省略了一些 C 代码。更完整的实现源代码可在线获取。[1]

我们先从程序的核心——控制器本身开始。图 2b 的第一行标明了目标语言

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



(a) 结构概览图

```
1 target C语言; 2 reactor 控制器 { 3 输入 angles:float[]; 4 输出 control:float;
5 初始模式 摆起 { 6 反应(angles) ->control, 重置(捕获) {=7 ... 此处为 C语言控制
律 ... 8 lf_set(control, ... 控制值 ... ); 9 if ( ... condition ... ) { lf_set_模式(捕获); }
10          =} 11 } 12 模式 捕获 { 13 反应(angles) ->control, 重置(稳定) {=14 ... 此
处为 C语言控制律 ... 15 lf_set(control, ... 控制值 ... ); 16 if ( ... condition ... ) { lf_
set_模式(稳定); }17          =} 18 } 19 模式 稳定 { 20 反应(angles) ->  control, 重
置(摆起) {=21 ... 此处为 C语言控制律 ... 22 lf_set(control, ... 控制值 ... ); 23 if ( ...
condition ... ) { lf_set_模式(摆起); }24          =} 25 } 26 }
```

(b) 控制器 反应器代码

```
1 目标C语言;
2 导入 Controllerfrom"FurutaPendulumController.lf"
3 导入 Sensor,Actuatorfrom"FurutaPendulumUtil.lf"
4
5 主反应器 {
6         s  =new 传感器();
7         c  =new 控制器();
8         a  =new 执行器();
9         s.角度 ->  c.角度;
10        c.控制 ->  a.控制;
11  }
```

(c) 程序的主反应器代码

图2. 驱动 FurutaPendulum 的 LF 程序。

如 C语言，这意味着该控制器将被转换为独立的 C 模块，并且反应和模式转换的逻辑将用 C 编写。控制器 反应器的前两行定义了输入和输出端口。它有一个名为角度 的向量值输入端口，用于接收包含轴角度、摆锤角度以及由传感器测得的两者角速度的测量值。接下来是三个反应 定义，它们各自对角度 输入作出反应，生成控制输出，并实现三种控制律之一。

我们在这里使用新的模式扩展，将每个一封装在

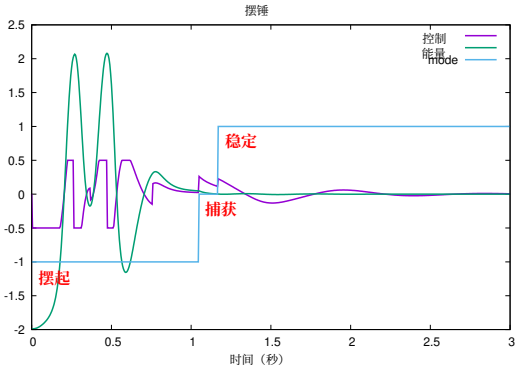


图 3. 使用控制器 对模拟摆锤进行控制的行为曲线。

单独的模式中。反应主体采用普通的 C 代码编写，用于读取输入值并计算驱动信号。这些 C 代码本应位于第 7、14 和 21 行，但由于这些细节与本文无关，此处略去。第 8、15 和 22 行将计算出的控制值发送到输出端口。第 9、16 和 23 行使用 C 表达式（此处略去）来判断是否需要立即进行模式切换；如果需要，则调用 `lf_set_模式` 以指定下一个模式。有关这些模式转换的语义，请参见第 IV-B 节。

接下来，图 2c 展示了如何在实际程序中使用该控制器。该程序会被编译为独立 C 程序，可加载到嵌入式控制器的闪存中并部署。为了与现实世界交互，还需要实现传感器和 执行器 反应器。为简洁起见，我们不再介绍它们的实现，只像控制器那样导入它们。第 5 行的主反应器代表实际程序，它实例化这两个反应器以及导入的控制器，然后连接它们的端口（第 9 行和第 10 行）。

我们采用 Eker 等人给出的模型 [19]，构建了一个实际摆锤的简单前向欧拉仿真，并使用我们的控制器运行该仿真。结果如图 3 所示。该图显示，控制器在摆起 模式下运行了一秒多一点，在捕获 模式下运行了约 100 毫秒，然后保持在稳定模式。

我们承认，这个控制器相当简单，即使不采用本文提出的模式扩展，也很容易实现。图 4 中勾勒了这种实现方式。这个变体以一段目标语言代码作为前言，用于定义一个枚举；随后，在响应传感器输入的反应主体中，使用 if-then-else 语句来确定要调用哪条控制律。对于这个示例而言，不显式使用模式的版本并非不可取；但对于更复杂的系统，我们认为，图 2 中的模态版本模块化程度更高，也更容易理解。此外，图 4a 依赖于手写的状态机实现。虽然对于仅有三个状态的情况，这或许可以接受，但它违背了模型驱动工程的基本理念，而且容易出错、难以扩展，也妨碍形式化验证。从建模角度来看，显式使用模式还能带来更

```
1  目标 C语言；2 前置声明 {=3 typedef enum {摆起, 捕获, 稳定} 模式;4  =} 5 反应器 控制器 { 6 输入 角度:double[]; 7 输出 控制:double; 8 状态 my_模式:模式; 9 10 反应(角度) ->控制 {
11                          >          ==          if (self- my_模式 摆起) { 12 ... 此处为 C语言控制律 ... 13 lf_set(控制, ... 控制值 ... ); 14 if ( ... 条件 ... ) { 15 self->my_模式  =  捕获; 16 } 17 } else if (self->my_模式  ==  捕获) { 18 ... 此处为 C语言控制律 ... 19 lf_set(控制, ... 控制值 ... ); 20 if ( ... 条件 ... ) { 21 self->my_模式  =稳定; 22 } 23 } else { 24 ... 此处为 C语言控制律 ... 25 lf_set(控制, ... 控制值 ... ); 26 if ( ... 条件 ... ) { 27 self->my_模式  =摆起; 28 } 29 }
30          =} 31 } (a) LF代码 (b) 图
```

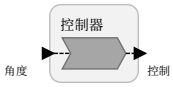


图4. 不使用模态模型实现控制器 。

有意义的图，将图4b与图2a进行比较。在更高级的模态场景中，模式局部时间和历史转换（稍后解释）等特性增强了表达能力，而这些能力在非模态 LF 中很难复制。

III. 面向反应器的编程

LF 协调语言的核心范式——面向反应器的编程——直到最近才被引入 [21]。它主要基于从成熟范式中借鉴的原则（例如面向对象 [30]，、基于参与者 [13]，、事件驱动 [8]，、基于流 [7],等）。

为了说明为什么我们的贡献是扩展反应器，而不是其他计算模型，我们首先讨论反应器模型的关键原则及其优势；本文其余部分将以此为基础。已经熟悉这些内容的读者可以跳过本节。反之，对于有兴趣了解更多细节（包括形式化定义）的读者，我们请他们参阅其他资料 [20], [22]。

A. 可组合性与因果关系

反应器在许多方面类似于对象；它们以可实例化类的形式声明，并提供一种继承形式。与对象一样，反应器封装状态；反应器中的所有状态变量均受保护，对子类可见，但除此之外不可见。对于对象而言，交互主要通过方法调用进行。相比之下，反应器具有反应、端口、计时器和动作，其中 反应是一种过程，会响应输入、计时器或动作事件而被调用。与方法不同，

反应不会被直接调用，而是由事件的出现触发。

端口是用于在反应器之间传递事件的终端。不同反应器的端口可以通过连接相连，从而实现事件流动。动作与端口非常相似，但它们是私有的，仅用于调度未来的事件（而端口用于传递当前已存在的事件）。动作有两种：物理动作，用于从中断服务例程等异步上下文中进行调度；以及逻辑动作，只能在反应执行过程中进行调度。通过逻辑动作调度的事件相对于当前标签存在偏移，而通过物理动作调度的事件则相对于物理时钟的测量值存在偏移。计时器是会按照预定义周期自动重新调度的动作。

每个反应都指定了触发器——可以触发该反应的端口、动作或计时器；源——触发时可供该反应读取的端口；以及效果——该反应可能在其上产生事件的端口或动作。作用域规则确保反应签名中表达的数据依赖关系是保守的。因此，反应签名编码了一个因果接口 [18]。

因此，反应器的组合可以转化为依赖关系图，将所有反应组织成偏序，其中包含必须遵守的所有调度约束，以确保反应器程序的执行产生确定性结果。由于无论触发反应时执行的代码内容如何，这个图都有效，因此可以将反应视为黑盒。正是反应器的这一特性使 LF 具有多语言特性。

B. 反应性与同步性

只有在存在触发反应的事件时，控制权才会交给反应。其背后的运行时机制可以与 JavaScript 的事件循环 [24] 相比较。不过，JavaScript 事件循环是异步的，而反应器的执行模型更接近同步语言 [3], ，因为执行会从一个同步反应式“时钟周期”推进到下一个。时钟周期对应单调递增的标签，只有标签匹配的事件才会被视为在特定时钟周期内存在。在给定时钟周期产生的输出在逻辑上是瞬时的，因此可能会触发更多反应在同一时钟周期发生。重要的是，运行时系统会确保在反应所依赖的所有值都已确定之前，不会执行该反应。这正是反应器具有确定性语义的原因。

C. 时间与并发

事件标签 划定了逻辑时间线，同时也用于使系统与物理时间保持同步。标签是由 (t, m) 组成的对，其中 t 是时间值， m 是微步索引；后者仅用于允许后续时钟周期在时间不流逝的情况下发生，这一概念称为超稠密时间 [25]。当最早可用事件标签的时间值大于当前物理时间时，执行引擎便会启动一个新的时钟周期。这一策略确保通过物理动作（来自异步执行线程）安排的事件可以标记为当前物理时间，而不会产生事件标签顺序错乱的风险。反应器模型中物理时间与逻辑时间之间的关系赋予了计时器实用的语义，也使截止时间的表述成为可能 [23]。这些构造对于在信息物理系统（如 FurutaPendulum）中运行的软件尤其有用。

由于反应器的执行由依赖图引导，因此在逻辑上同时发生且彼此之间没有依赖关系的反应，可以透明地并行执行，而不会引入数据竞争、死锁或其他多线程执行时常见的问题 [16]。目前，C、C++ 和 Rust 的运行时实现会自动利用这类并行性（前提是执行平台拥有多个处理器核心）。需要注意的是，同一反应器中的反应之间始终存在依赖关系，以确保对共享状态的互斥访问。具体而言，代码中的声明顺序决定了优先顺序。

D. 可视化与语用学

这种由组件、端口、连接和层次结构构成的面向反应器的编程范式，尤其适合采用可视化方式，帮助程序员理解代码。非基于流的编程模型可以进行控制流分析，并生成图形化呈现来帮助解释程序行为；但通常很难确定此类分析的合适范围或粒度，因此难以实现自动化。更常见的是能够展现高层软件架构（例如类图）的可视化方式；虽然这类方式易于自动化，却很难揭示程序的实际行为。

LF 内置自动交互式图示合成功能，旨在帮助开发者更好地理解代码并提高工作效率。可以说，在处理可能非常复杂的系统模型时，LF 善于运用“语用学” [33] ，着重探索如何充分发挥文本与图形两种形式的优势。本文介绍的扩展利用了这一功能，并增加了专为模态反应器设计的可视化。所有 LF 图（包括本文展示的图）均由 LF 代码自动合成，所用算法是由 Eclipse 布局内核（ELK）[2]提供的、支持端口的知名杉山算法变体 [29], 。

IV. 模态反应器

模态反应器的基本思想是沿用现有的反应器模型，同时通过将反应器划分为互不重叠的子集，并将这些子集与互斥的模式相关联，实现对反应的模态协调。在模态反应器中，每个逻辑时间瞬间只能有一个模式处于活动状态，这意味着其他模式中的活动会自动暂停。切换模式会改变反应器的行为，并控制进入模式的起始点。重置模式历史或继续沿用模式历史的选项，是常见且强大的抽象，尤其有助于管理

[2]https://www.eclipse.org/elk/

复杂的定时行为；如果完全依靠手动处理，这类行为极易出错。

尽管模态反应器背后的理念并不新颖，但 LF 的基本原则为多语言模态协调层提供的指导却是开创性的。我们的目标是创建具备以下特点的模态模型：

- **精简** 一个精简的协调层，在提供最基本功能的同时，仍具备最大的灵活性和用户可调节性；
- **多语言** 一个灵活的多语言封装器，专注于用户所使用的语言，且只需少量适配工作；
- **并发**支持设计多个彼此独立运行的模态单元；
- **定时** 以可靠且精确的方式指定对时间敏感的模态行为，即使是在并行和分布式环境中；并且
- **确定性**对于相同的带标签输入事件序列，产生明确且可复现的输出行为。

我们对 LF 的模态扩展体现了这些原则，并采用了至关重要的反应“黑盒”方法。

A. 语法

添加模式所需的语法改动相当少。新增语法允许将反应分组到模式中，并加入用于声明（初始）模式及指定转换类型的新关键字。LF 核心语言保持不变。

模式可以在任何反应器中定义。每个模式都需要一个唯一的名称（在所属反应器内唯一），并且可以声明仅属于该模式的内容。必须且只能有一个模式标记为初始（见图 2 第 5 行）。模式可以包含状态变量、计时器、动作、反应、反应器实例化和连接。虽然模式不能直接嵌套在其他模式中，但可以通过实例化模态反应器实现层次化组合。模式中允许包含的内容，唯一的主要例外是端口声明，因为端口声明只能出现在反应器级别。不过，模式与其反应器共享作用域，因此可以访问反应器的端口、状态变量和参数。其他模式中的内容则除外。

模式转换在反应中声明。如果反应器包含模式，则允许反应将这些模式列为效果。这样便可使用目标语言 API 设置下一个模式，方法是使用 `lf_set_mode` （另见图 2b 中的第 9 行）。如果目标代码引用了未声明为效果的模式，编译器将拒绝该程序。用户还必须通过在效果中添加修饰符`reset` 或 `history` 来指定转换类型。声明为 `history(<模式>)` 的效果表示到该模式的历史转换，在图形语法中以箭头尖端处的“H”表示（见图 5a）。

B. 模式与转换

LF 中模式的基本作用是，只有包含在当前活动模式中或不包含在任何模式中的部分，才会在任意时刻执行。由于层次结构的存在，这也适用于嵌套在多个 祖先模式 中的部分；因此，所有这些祖先模式都必须处于活动状态才能执行。非活动模式中的反应不会执行。所有用于建模定时行为的组件，即计时器、已调度动作和延迟连接，都遵循局部时间的概念。这意味着模式处于非活动状态时，局部时间的推进会暂停。模式激活时，定时组件的行为取决于转换类型。进入模式时可以对其进行重置 ，使其恢复到初始状态。或者，如果该模式之前处于活动状态，也可以根据其历史继续运行。在后一种情况下，所有定时组件都会继续计时其延迟或周期，就像模式处于非活动状态期间时间并未流逝一样。第 IV-D 节将进一步阐述局部时间的概念。

反应器启动时，每个模态反应器的初始模式处于激活状态，其他模式均未激活。如果在某个标签处（$t,\ m$），该反应器及其所有内容中的反应均已执行完毕，并且某个反应设置了新模式，那么当前模式将被停用，新模式将在后续执行中激活。这意味着，新激活模式中的任何反应都不会在标签处（$t,\ m$）执行；新模式中最早可能发生的反应要到一个微步之后，即在（$t,\ m+1$）才会执行。由于采用超稠密时间模型，LF 能够在同一逻辑时间上建模后续反应，只是晚一个微步。因此，举例来说，如果新激活的模式中有一个偏移量为零的计时器，它将在（$t,\ m+1$）触发。如果模式本身不需要在下一个微步立即执行，那么它就依赖于未来事件，这与 LF 的常规行为相同。因此，在超稠密时间意义上，同一反应器中的模式始终互斥。第 IV-C 节将进一步讨论这一设计。

如果在反应体中设置了新模式，就会触发转换，如图 2b 第 9、16 和 23 行所示。与在反应中设置输出端口一样，同一反应或不同反应中都可以多次设置新模式。最终，由反应的固定顺序决定将采用最后一个有效值。新模式不必与当前模式不同；模式可以通过重置转换来重置自身。

如果以重置行为进入某个模式，则其中包含的所有模态反应器都会重置为各自的初始模式（递归进行），所有局部计时器都会重置并重新开始等待其初始偏移量，先前在此模式内安排的所有事件（动作、计时器、延迟连接）都会被丢弃，并且新引入的 重置 触发器会激活该模式及其所有包含的反应器（递归进行）中的相关反应。

因此，每当通过重置转换进入某个模式时，后续的定时行为都如同该模式此前从未执行过。默认情况下，状态变量不会自动重置，因为在反应器中，惯常做法是在由启动触发的反应中分配资源或初始化子系统（例如，分配内存或套接字、注册中断或启动服务器），并将这些资源的引用存储在状态变量中。因此，如果在模式内部使用启动和关闭反应，则必须对其进行特殊处理，以确保模式活动期间资源得到正确的

分配和释放。第 IV-E 节将详细讨论这一主题。无论如何，如果某些状态变量需要重置或重新初始化，可以在由重置触发的反应中完成，也可以在其声明上标记重置，从而自动赋予其初始值。

另一方面，如果某个模式之前曾处于激活状态，随后又通过历史转换重新进入，则不会执行重置。计时器、已调度动作和延迟连接产生的事件会进行调整，使其剩余延迟等于该模式此前停用时记录的剩余延迟。因此，模式具有局部时间的概念：只有在模式处于激活状态时，局部时间才会流逝，参见第 IV-D 节。

C. 转换时序

LF 中关于模式的一项重要设计决策涉及转换的时序，也就是转换究竟在何时发生。已有多种成熟的语义，两个基本选项是“立即”转换，或“延迟”转换：在后一种情况下，目标模式至少要等到发起转换后的一个微步才会被启用。

若转换效应立即生效，转换就可能在执行发起它的反应后直接发生，从而立刻激活目标模式中的反应。这意味着，带有转换效应的反应与目标模式中的反应之间存在依赖关系。这类额外依赖可能会在原本合法的模态模型中引发因果循环，并削弱可利用的并行性。另一种转换时序方案是，等到某个模式的所有内容执行完毕后，立即切换到下一个模式，并在同一标签下执行该模式。然而，这会引出一个问题：如何处理同一模式在同一标签下多次重新激活的情况。此外，这还会允许在同一执行时刻发生任意多次模式切换。归根结底，模式激活之间需要某种顺序关系；而在我们采用的延迟方案中，微步本质上正是用于此目的。

在 LF 中，反应可以设置一个新模式，但这不会立即生效。只有当反应器执行完其所有内容后，转换才会执行，新模式也会在下一个微步变为活动状态。采用这种设计，同一反应器中的两个模式不会在同一标签处同时处于活动状态，转换也不会干扰正在进行的反应。然而，这种方法需要解决不同反应所产生的潜在“冲突”转换效果。在这里，我们采用 LF 中设置端口时已使用的相同机制，并依据反应器内部反应的固定顺序来确定有效的目标模式。就确定性结果和覆盖行为而言，设置新模式可以视为类似于赋值给输出端口。不过，就时间安排而言，转换效果相当于调度一个零延迟动作，这也会强制引入一个微步延迟，以防止因果循环。

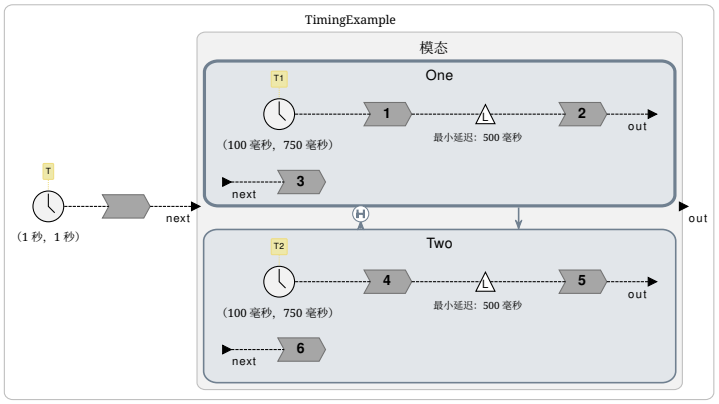
我们选择的设计类似于 SCADE [6], 中的设计选择：一个反应中的状态彼此互斥。它不同于 SyncCharts [2]中的设计选择或 SCCharts [32], 中的设计选择：在一次反应中，多个状态可以通过一系列“即时转换”同时激活。请注意，这两种选择都能实现确定性语义。对于 SCCharts，确定性是通过预先确定的状态顺序排列来确保的。

D. 局部时间

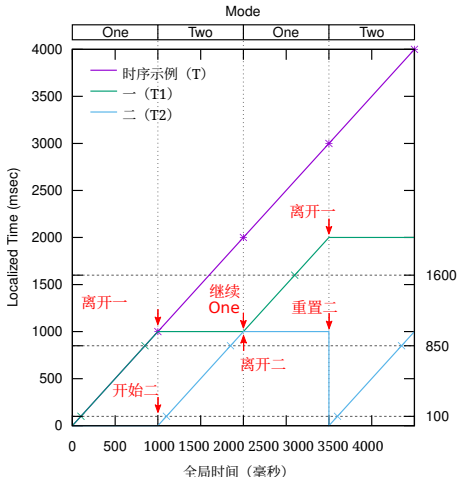
模式局部时间这一概念，即暂停非活动模式中的所有定时行为，是一项成熟且定义完善的原则，在 Ptolemy II 的模态模型中也有应用 [17]。其中支持局部时间而非其他替代方法的考量同样适用于 LF。暂停时间为模式的非活动状态赋予了清晰、一致的含义，也使进入模式时其内容所处的状态易于理解。这尤其有利于模块化，因为可能在模式中实例化的反应器无需预先考虑：当其反应被抑制时，其时间（由计时器或已调度动作驱动）仍会继续推进。此外，模式还允许在模式之外定义反应器元素，使开发者能够控制时间是否应局限于某个模式。

图 5 展示了在两种转换类型下，局部时间对计时器和动作产生的不同影响。图 5a 显示了为一个合成示例程序生成的图。它由两个模式一（初始模式）和二组成，两者都属于模态 反应器。下一个 输入用于在这两个模式之间切换。输入端口由顶层的一个反应控制，该反应由计时器T触发。一秒后，模式切换开始以一秒为周期定期触发。两个模式的内容结构相同。每个模式都有一个计时器T1/T2 ，它在初始偏移量 100 毫秒后触发一个反应，之后每隔 750 毫秒定期触发一次。随后，该反应会安排一个延迟 500 毫秒的逻辑动作（实际目标代码不会在规定的最短延迟之外再增加延迟）。此动作触发第二个反应，该反应向输出输出写入数据。最后一个反应由下一个 输入触发，并调用转换以切换到另一个状态。两个模式的主要区别在于，进入一 时会经过历史转换，从而延续其行为；而二 则会重置。

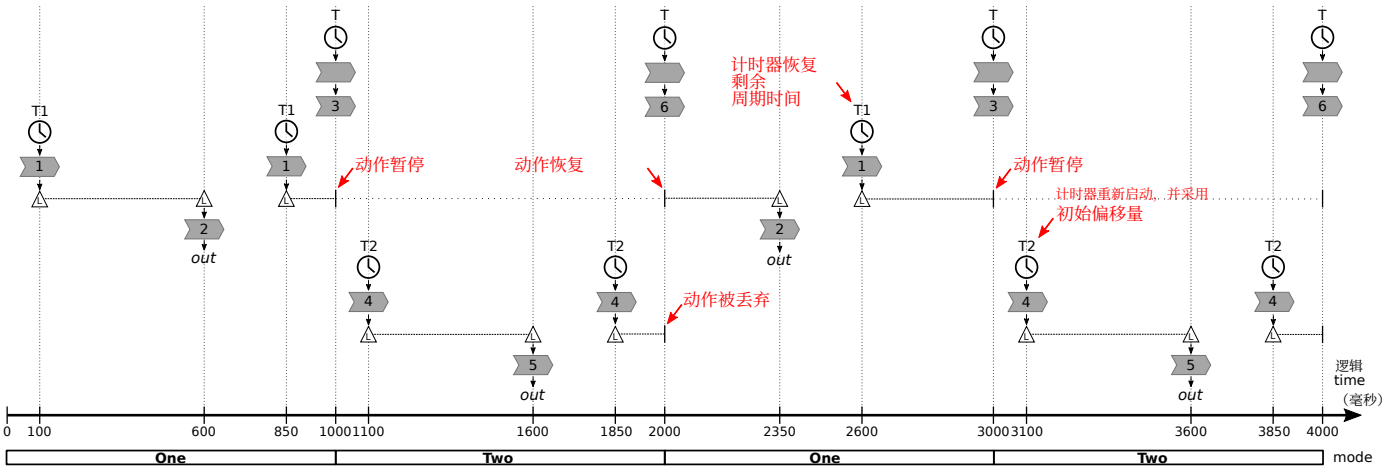
图 5c 展示了该程序前 4 秒的执行轨迹。时间线下方显示当前激活的模式，时间线上方则显示在特定时刻执行的模型元素，并用箭头表示触发关系，用虚线表示时间上的分布。例如，在 100 毫秒时，计时器 T1 的初始偏移量到期，从而触发对此模式中逻辑动作的调度。该动作在 500 毫秒后，也就是 600 毫秒时，触发反应并产生输出。时序图展示了历史转换与重置转换在时间处理上的不同。具体来说，在 2000 毫秒时，通过历史转换重新进入模式一 时，此前在 850 毫秒由T1 触发的动作会恢复。相比之下，在 3000 毫秒时，通过重置转换重新进入模式 二 时，此前在 1850 毫秒由 T2 触发的动作会被丢弃。



(a) LF 模型



(b) 各模式中的时间推进及其各自的 计时器



(c) 带有反应示意的执行轨迹

图 5. LF 示例展示了 重置和历史转换对计时器产生的不同效果 以及模式中的延迟。

图 5b 展示了环境中的全局时间与图 5 中每个计时器的局部时间之间的关系。由于顶层反应器 TimingExample 不被任何模式包含，其时间始终与全局时间一致。模式 一 是初始模式，因此在最初的一秒内与 TimingExample 同步推进。模式 一 未激活期间，计时器会暂停，不再推进时间。在 2000 毫秒时，它会从此时继续计时。T2 只有在模式于 1000 毫秒变为激活状态时才开始计时。在 3000 毫秒时通过重置重新进入该模式，会使局部时间重置为零。

简而言之，从计时器和动作的角度来看，模式处于非活动状态时，时间会暂停。这也适用于模式中间接嵌套的反应器。同样，如果逻辑延迟连接的源位于某个模式内，那么此类连接（在 LF 中由 `after`关键字指定）也会受到局部时间的影响。这是因为延迟连接在一定程度上是通过逻辑动作实现延迟的连接的一种语法糖。

E. 启动与关闭

我们面临的一项设计挑战是如何处理程序执行的开始和结束。LF 定义了两个特殊触发器：`startup` 和 `shutdown`。由 `startup` 触发的反应会在反应器存在期间的第一个标签处执行，而由 `shutdown` 触发的反应则会在反应器不复存在前的最后一个标签处执行。它们通常用于初始化反应器，例如为状态变量分配内存、初始化传感器或执行器，或启动外部线程（例如，用于监听网络输入）。`shutdown` 反应随后会在退出前进行清理。如果在某个模式内发现了此类反应，该怎么办？进入或退出模式时是否应该执行这些反应？

我们考察了几种不同的方案，最终得出结论：每种方案都有一些古怪之处和特殊边界情况。我们采用了一种易于说明、结果确定的策略，并为我们所能识别的所有古怪之处提供了变通方法，但我们承认对此仍有些不满意。我们的解决方案是一种折中。

首先，我们在某个模式的第一个激活标签处，最多调用一次启动 反应

。其次，在所属反应器即将停止存在之前（通常是在程序退出前），我们会调用关闭反应，这与模式无关；但我们只会在至少激活过一次的模式中调用这些反应。因此，每个启动 都有相应的关闭。第三，如前所述，我们引入了一种新的反应重置 触发器。由重置 触发的反应会在每次通过重置转换进入某个模式时执行。

例如，可以考虑内存分配和释放。现在可以在启动和关闭 反应中安全地在模式内执行这些操作，不过这种策略的缺点是，模式处于非活动状态时不会释放内存。若要分配仅在模式处于活动状态时使用的内存，程序员可以在重置 反应中分配内存，并在任何调用`lf_set_模式` 以退出该模式的反应中释放内存。然后，程序员必须确保每个进入转换都是重置转换。

我们的设计带来一个奇特之处：即使在非活动模式中，也会调用关闭反应。通常，不应调用非活动模式的任何反应。这是唯一的例外，但它确保每次触发启动 反应时，都会有相应的关闭反应被触发。

另一种方案是在离开模式时执行关闭反应，但这会使历史转换无法实现。我们还考虑过将关闭 反应分散到多个连续的微步中，使任意两个不同模式的反应在逻辑上都不会同时发生。这种做法似乎难以解释，而且可能会导致更多奇特的现象。我们考虑的第三种可能性是在开始时执行所有启动反应，不论相应模式是否处于活动状态。但这样一来，启动 反应就会和关闭 反应一样奇怪，而我们认为，奇怪的行为还是越少越好。

F. 模式图

如第III-D节所述，我们认为（自动）绘图是 LF 的一个重要方面，也将模态行为作为这种图形表示法中的一等公民。即使模式切换的实际触发发生在逐字目标代码中，而不属于 LF 语言本身，但只要在反应接口中声明模式和可能的转换，就足以向用户提供模式图。不过，用户可以在 LF 层面指定宿主代码中实际上不存在的状态转换（但反之则不行，正如第IV-A节所述）。因此，模式图是保守的：它们展示了所有可能的行为，但也可能显示宿主代码中并未实现的转换。

对于 LF 图，一个重要的设计决策是如何在视觉上呈现数据流与控制流之间的关系。在 LF 中，反应器以圆角矩形表示，其数据流则通过矩形边缘布线呈现。遵循状态机模型的既有惯例，模式也以圆角矩形表示，但采用不同的配色方案。初始状态用较粗的轮廓线标示。图2a展示了这种表示法。状态转换以样条曲线绘制。对于在模式中继续行为的转换如图 5a 所示，我们在箭头前添加一个带圈的“H”，表示进入该模式时将恢复其先前的历史。

与本文中的其他 LF 图不同，图2a中的转换还带有标签。LF 图提供了多种方式，让用户影响图的外观和细节程度。因此，模式也有一些配置选项，例如是否显示标签。标签能够指出哪些输入和/或动作可能导致模式转换，从而提供额外价值。同样，是否添加标签也需要考虑一些设计因素。传统上，转换标签包括触发器和效果。然而，这需要分析宿主代码，与多语言方法相悖；而且，实际导致转换发生的条件，以及转换发生后产生的效果，都可能变得任意复杂。因此，我们选择将转换标签限制为可能 触发反应的事件，而省略宿主代码内部决定转换是否实际发生的其他逻辑。如果配置为不显示标签，且因此没有任何可用于区分转换的因素，我们就会将相同模式之间的多个转换合并为一个。

另一个不太明显的问题是，如何将反应依赖关系整合到新的模式和转换层中。一种做法是尝试混合所有这些边。但这意味着数据流边可能需要跨越模式的层级，才能连接模式中的任意内容。我们为此制作了几种视觉模型图。所有包含某种形式的跨层级边的方案，一旦规模超过简单示例，就显得相当杂乱。此外，我们还考虑了折叠模式以隐藏其内容时的交互方式，以及这些方案与自动布局算法的兼容性。最终，我们选择在每个模式所在的层级拆分这些外部连接。这样会让某些连接不那么直观，但能让图更简洁，也能简化布局问题。对于端口，我们会复制模式中使用的端口，并用其箭头图形表示。端口名称用于与原始反应器端口建立关联。在图2a中，你可以看到这种方式应用于触发反应的角度 输入。对于连接到模式外部定义的反应器元素（例如计时器）的连接，我们会创建人工类端口元素来表示它们之间的关系。

五、实现

创建侵入性最小的模式扩展这一目标，不仅体现在语言设计本身，也体现在运行时实现中。LF 采用多语言方法，支持多种目标语言。撰写本文时，模式已在 C语言和 Python 中实现，未来也已计划扩展支持范围。尽管如此，这些模式已在这些目标语言上成功通过单线程、多线程和分布式执行测试。在本节余下部分，我们将概述支持模式所需的调整，力求提供通用且合理独立于目标语言的视角，以实现

第 IV 节所述的行为。实际实现可在 GitHub 上的 LF 项目中找到。[3]

LF 生成的代码由两部分组成。第一部分是特定于程序的部分，用于表示程序本身的组件和拓扑结构，例如反应器、反应器的实际实例化以及它们之间的连接。第二部分是通用运行时环境，负责处理事件队列中的事件并进行调度，以及触发和执行反应器的反应。为支持模式而对这两部分进行的调整相对较小。此外，这些调整大多是增量式的，这意味着如果源模型中不存在模式，那么与不考虑模式的实现几乎没有差异[4]。

现有的运行时实现需要进行两方面的调整。第一，触发反应时必须检查该反应是否处于活动模式中，否则就阻止其执行。一种简单的方法是递归检查某个模式及其所有父模式，确认它们是否都是当前活动模式；而与任何模式都无关的元素则视为始终处于活动状态。关闭反应不参与此活动状态检查，但其所属模式（如果有）必须已经经历过启动阶段；执行这些反应时也会额外记录这一信息。第二，执行生命周期需要处理转换。处理反应完毕后、逻辑时间推进之前，必须调用转换处理。

算法 1 展示了处理模式转换的过程，其中包括执行重置、管理局部时间以及调度特殊触发器。该算法依赖于一些全局数据结构：(1) LF 现有的事件队列（事件队列），对其进行操作是实现局部时间所需的唯一运行时改动；(2) 用于存储在局部时间中挂起的事件的集合（挂起事件）；以及 (3) 模型中所有模态反应器实例的集合（模态反应器）。后者由程序专用生成代码以及一些新的数据结构和引用构成。每个模态反应器实例r 都可访问以下信息（此处以成员表示法呈现）。 r_{modes} 表示 r中的模式集合。

r.父模式 $\in \{\text{Nil}\} \cup \{x_{modes} : x \in$ 模态反应器$\}$ r.初始模式 $\in r_{modes}$ r.当前模式 $\in r_{modes}$ r.下一模式 $\in \{\text{Nil}\} \cup r_{modes}$ r.转换 $\in \{\text{None, Reset, History}\}$

r.parentMode 要么不存在，要么是直接包含 r 的模式。请注意，这里仅对模式层次结构进行单向建模。虽然也可以考虑引入一个列表来记录所包含的模态反应器，但我们保留了这种表示方式，以贴近实际实现，尽管这会对算法的结构产生影响。 r.initialMode 是必需的初始模式。与父模式一样，这两个值都是常量，并在程序启动时设置。 r.currentMode 是相对于 r 当前处于活动状态的模式，初始值为初始模式。当前模式是否在执行过程中实际处于活动状态，还取决于父模式。 r.nextMode 和 r.transition 表示转换是否存在、转换类型及其目标。如果目标代码设置了新模式，就会填充这些字段，例如图 2b 第 9 行。实际上，转换类型是根据效果定义推断出来的。此外，每个模式 m in r_{modes} 都包含额外的模式特定信息。

m.reactor $\in$模态反应器m. leaveTime $\in \mathbb{T}$ m.reset $\in \{$ True，False$\}$ m.hadStartup $\in \{$ True，False$\}$

m.reactor 是对该模式的反应器的常量引用。m.leaveTime 存储该模式最近一次退出时的逻辑时间。它使用执行的开始时间进行初始化。m.reset 表示该模式一旦激活就需要重置（初始值为 False）。 m.hadStartup 是一个布尔标志，该模式首次激活时会立即从 False 设为 True。最后，每个反应、计时器和动作也都引用其直接外层模式（如果存在）。这样就能通过触发器将事件与模式关联起来。

在算法 1 的第一行中，按照自顶向下顺序处理每个模态反应器实例。这指的是模式层次结构的偏序，并确保进入某个模式时若执行了重置，其内部的模态反应器（第 3 行）会递归地通过重置转换进入初始状态。之后，在单独的一轮迭代中处理转换。之所以将这一轮迭代与上一轮分开，是因为层次化重置依赖父模式中的转换信息来相应地重置自身（第 3 行），而这些信息此时已被覆盖（第 31 行）。首先，处理下一个模式中那些因时间暂停的事件。发生重置时，所有计时器都会以相对于当前时间的初始偏移量重新启动（第 15 行）。其他事件（例如已调度动作）则会被丢弃。为将先前暂停的事件重新引入事件队列，使用移位函数创建相对于超稠密时间的正确标签：

$$\text{shift}(\text{base} : (t, m), \text{offset} : (t, m)) = \begin{cases} \text{offset}_t > 0 : & (\text{base}_t + \text{offset}_t, \text{offset}_m) \\ \text{offset}_t = 0 : & (\text{base}_t, \text{base}_m + \text{offset}_m + 1) \end{cases}$$

这会创建一个标签，即将基础标签向未来移位给定的偏移量。它会考虑到零延迟偏移量（相对于时间戳 t）会产生一个未来的（递增的）微步。如果由于历史转换而需要继续时间（从第 18 行开始），则所有事件都会以调整后的目标时间重新加入事件队列。这里，会从事件的原始标签（发生时间）中减去离开模式的时间，以得到离开时的剩余时间，并用它将该事件相对于当前时间进行偏移（第 19 行）。接下来，将转换的实际效果应用于内部数据结构（第 25 至 32 行）。这包括在必要时将模式标记为需要重置、存储离开该模式的时间、设置新模式，以及清除供未来执行使用的转换信息。之后，会触发当前模式的特殊反应。请注意，这一操作基于模式活动生效，并非由转换触发（第 33 行）。isActive 函数依赖于

[3]https://github.com/lf-lang [4]某些数据结构需要引用可能存在的外层模式。这是为了能够在模态反应器中使用预编译反应器。不过，这只会产生极少的内存开销，即一些在没有模式时不会使用的指针。

算法 1每个执行周期结束时对模式转换的处理。

```
1: 对每个 r ∈ 模态反应器自顶向下 执行
2:   如果 r.parentMode ≠ Nil 并且
3:         r.parentMode.反应器.转换 = 重置然后
4:     r.下一模式 := r.initalMode
5:     r.transition := 重置
6:   结束如果
7: 结束循环
8: 对于每个 r ∈ 模态反应器 执行
9:   如果 r.转换 ≠ 无 则
10:    对于每个 e ∈ 挂起事件 执行
11:      如果 e.模式 = r.下一模式 则
12:        从挂起事件中移除 e
13:        如果 r.转换 = 重置那么
14:          如果 e 是计时器 那么
15:            t:= 移位(currentLTime, e.计时器.offset)
16:            将 e 插入事件队列，标签为 t
17:          结束如果
18:        else
19:          t:= 移位(currentLTime,
20:            e.标签 - e.模式.leaveTime)
21:          将 e 插入事件队列，标签为 t
22:        结束 if
23:      结束 if
24:    结束 for
25:    如果 r.转换 =重置 那么
26:      r.下一模式.重置 := 真
27:    结束如果
28:    r.当前模式.离开时间 := 当前L时间
29:    r.当前模式 := r.下一模式
30:    r.下一模式 := 空
31:    r.转换 := 无
32:  结束条件
33:  如果isActive(e.currentMode) 那么
34:    如果未 r.currentMode.hadStartup那么
35:      触发启动 r.currentMode 中的反应
36:        在下一个微步
37:    结束 if
38:    如果 r.currentMode.reset则
39:      r.currentMode.重置 := False
40:      触发 r.currentMode 中的重置 反应
41:        在下一个微步
42:      重置 r.currentMode 中的状态变量，这些变量
43:        标记为 重置
44:    结束 if
45:  结束 if
46: 结束 for
47: 遍历 e ∈ 事件队列 执行
48:   如果非isActive(e.模式) 则
49:     从事件队列中移除 e
50:     将 e 添加到挂起事件中
51:   结束 if
52: 结束 for
```

前面给出的定义。如果该模式之前从未处于活动状态，其启动反应将在下一个微步触发。这包括该模式中的反应以及所有内部非模态反应器中的反应。同样，当模式被标记为需要重置时，会触发重置反应。此外，还会调用相应状态变量的自动重置，并清除该标志。最后，从事件队列中取出所有与当前非活动模式相关联的事件，并将其存储在挂起事件集合中（第 47–52 行）。

在算法 1 的实际实现中，该过程还包括一些额外的一致性检查、优化措施，以及用于处理一些次要边界情况的机制。不过，由于这些内容与整体语义无关，为了便于阅读，我们在此将其省略。总体而言，该算法相当紧凑，同时为模式提供了大部分语义功能。

VI. 讨论

A. 强抢占与弱抢占

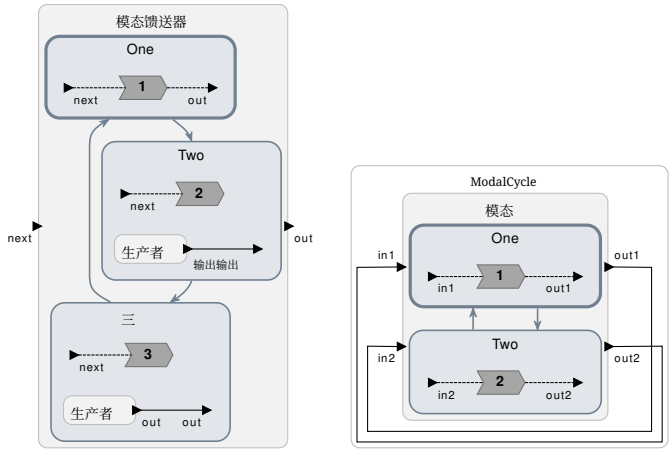
模式转换通常意味着抢占与被离开模式相关联的某些行为。同步状态图方言（如 SyncCharts 和 SCCharts）区分强抢占和弱抢占：前者表示发生抢占的时钟周期内不再执行此类内部行为，后者则允许模式在该时钟周期内执行“最后一个愿望”。因此，例如在 SyncCharts 中，如果某个状态能够发出信号并强抢占该状态，就会构成编译时因果关系错误。

从技术上讲，模态反应器中的转换是弱抢占的，也就是说，即使发生模式切换，所有反应仍会在当前时钟周期内执行。同步语义下，LF 层面没有强抢占的一个理由是，决定向其他模式转换的是模式本身；若采用强抢占，就会导致前文所述的因果性问题。我们曾考虑过添加强抢占的一些方案，例如引入特殊的“初始”反应，在反应开始时判断是否应该发生抢占，并可能抑制后续反应。不过，我们很快意识到，LF 无需任何扩展便已具备这一能力。是否进行转换由目标语言代码决定，程序员可以尽早作出决定，并阻止该模式中的后续计算。因此，模态反应器提供的转换已经能够实现这两种抢占行为。

B. 转换类型

第 IV-D 节已讨论了我们为何选择在 LF 层支持延迟转换，而不支持即时转换。一些语言（例如 SCADE 或 SCCharts）还提供一种延后 转换，用于抑制进入某状态后立即触发的反应；这与非延后转换不同，后者会立即激活目标状态的行为。由于模态反应器永远不会在同一个微步中执行目标状态，因此有人可能会认为我们的转换始终是延后的。

Statecharts 方言通常使用优先级 为可用转换指定顺序，优先级较高的转换会抢占较低优先级的转换。LF 中的模式行为恰好相反，



（a）包含多个馈送源的示例（反应和两个反应器）连接到同一个输出端口（输出），但由不同的模式区分。

（b）包含循环依赖的示例通过使用模式来解决。

图 6. 两个使用模式后可被视为确定性/因果的 LF 模型示例。

其中，最后 一次调用`lfsetmode` 决定_ _目标模式。很容易实现一种优先采用第一次调用`lf_set_mode`的实现方式，但我们更倾向于采用与端口行为一致的语义。

最后，一些状态图方言还区分深层和浅层历史转换，两者的区别在于其影响是否会沿层次结构向下传播。回顾我们在第 IV-B 节以及算法 1 中介绍的转换语义，可以看出，我们目前采用的是深层变体。我们曾简要考虑为模态反应器添加浅层变体，但认为由此带来的额外复杂性并不值得。再次强调，如果确实需要这种行为，仍可在目标语言层面实现。

C. 模式与因果关系

如前所述，重置和继续转换的可用性提供了对行为的时间维度进行控制的能力，而这种能力难以通过简单的代码内变通方法实现。通过模式在结构上表达互斥的可能性也是如此。这突破了标准 LF 计算模型所施加的某些建模限制。例如，在没有模式的情况下，一个输出端口不得通过多个连接连接到反应器，也不得同时连接反应和反应器，因为这可能导致非确定性。模式允许采用此类配置，前提是每个写入方都位于单独的模式中，以确保输出之间互斥（示例见图 6a）。

同样的推理也适用于反馈环引发的因果性问题。模式的使用能够实现更细粒度、保守性更低的依赖分析。例如，若没有模式，图 6b 中的程序会因以下因果循环而被拒绝：in1– out1 –in2– out2–in1。两种模式互斥，因此消除了这一因果循环。

D. 作为变更的模式

如本文所述，模式被设计为 LF 的核心语言特性。最初，我们也曾考虑基于变更来实现模式；变更已属于反应器形式化定义的一部分 [21]。然而，尽管在一定程度上可以这样做，但从方法论角度考虑，这种做法并不可取。首先，变更旨在于运行时重塑 LF 的拓扑结构，而模式围绕活动与非活动的概念构建，因此始终静态存在。其次，目前通过变更创建和销毁反应器时，不会考虑任何先前状态。如果希望实现继续行为，那么将转换类比为销毁旧模式并创建新模式，就会变得相当复杂。按照本文提出的方案，继续仅仅意味着不进行重置。

E. 模式的形式化分析

在语言中集成模式为验证和模型检查提供了新的机会，因为更多结构信息已在静态层面呈现。然而，当前设计的一个局限在于，转换触发位于反应代码内部。与反应上声明的其他效果一样，在 LF 层级声明的、指向某个目标模式的转换效应，只反映了产生此类效果的可能性。实际调用取决于反应代码的执行，而这又取决于是否存在任何触发器及其运行时值。因此，在不执行目标代码的情况下，无法真正对活动模式进行静态估计。不过，对于 LF 中任何依赖状态的验证而言，通常都存在这一问题。由于状态变量和端口表示目标语言中的值，因此对与目标语言无关的仿真存在隐含限制。因此，将潜在的转换目标声明为效果符合 LF 的多语言方法，并足以支持图生成和结构分析。目前，针对未来 LF 开发中包含模式的验证能力扩展，正在进行考虑。

七、相关工作

据我们所知，尚无其他工作旨在基于类反应器基础创建多语言模态协调层。不过，显然有大量相关工作构成了我们方案的基础。

同步建模语言 SCADE [6] 最初侧重于数据流，多个并发运行的节点通过流进行通信。不过，从 6 版开始，它也加入了状态机，其基础是 Colaco 等人提出的模式扩展 [5]。他们的工作或许在理念上最接近我们在此提出的方案，部分原因是他们通过最小化的语言扩展实现了模式。具体而言，他们以相当优雅的方式扩展了最初用于 Lustre/SCADE 中控制执行的布尔时钟，将其扩展为能够编码模式的更丰富类型。SCADE 是一种独立语言，虽然能够与其目标语言集成，并通过信号协调行为，但它并未体现 LF 所设想的严谨的多语言协调特性。

更广泛地说，除了扁平有限状态机（有限状态机）之外，还有许多状态机表示法提供了功能丰富的语言构造。其中最突出、功能最强大的是 Harel [12], 提出的 Statecharts，它实现了层次状态机，也是统一建模语言的一部分。在同步语言背景下发展起来的状态图方言，例如 SyncCharts [2] 或 SCCharts [32], ，与 LF 中的模式尤其相关，因为它们的语义也包含明确的时间概念和反应。有一些研究探讨如何将物理时间纳入同步计算模型，同时保持确定性 [9], [28]。不过，通常所谓的“多形式时间概念”是基于对事件的计数，例如经过 1 毫秒。

另一种常见做法是直接用传统编程语言表达状态图，而不进行真正的语言扩展，这与图 4 类似。Samek 描述了如何用 C/C++ [27] 表达 UML 状态图。与 UML 状态图一样，这种方法不提供确定性并发。Wagner 等人描述了如何用 C [34], 实现有限状态机，但这些都是不含并发的扁平自动机。此外，它们都没有采用多语言协调方法。

类似地，Akka 框架 [26] 提供了直接用 Java 编写参与者网络的方法，但没有考虑模式或确定性。

人们提出了各种方案，试图为 C 等主流编程语言增添状态或模式的概念。Esterel-C 语言（ECL） [15] 提议通过类似 Esterel 的结构扩展 C，以支持信号处理和反应式控制流；ECL 编译器则从该程序中生成一部分 Esterel 代码和一部分纯顺序 C 代码。FairThreads [4] 是另一种受 Esterel 启发的 C 扩展，通过原生线程实现，并提供用于表达自动机的宏。精确计时 C（PRET-C） [1], 着重于时间可预测性，并假定目标架构具有对线程调度和中止处理的特定支持；ForeC [35], 面向多核架构；二者都通过暂停语句为 C 程序引入模态行为。在这些面向 C 的同步扩展中，与我们的方案或许最接近的是 C 中的 SyncCharts：它通过轻量级语言扩展（以 C 宏实现）为 C 增添基于 SyncCharts [31] 的模式。这些宏通过使用粗粒度程序计数器控制执行来实现状态和并发；在技术上，这些程序计数器由普通 C 标签和计算跳转实现。通过按照（通常是静态的）优先级分派并发线程，实现确定性。

VIII. 结论与展望

模态反应器能够通过模式和转换来协调反应式行为。为了实现精简、多语言、并发、定时且确定性的目标，我们充分利用 LF 中许多现有概念，同时谨慎调整这些基本原则，使其无缝融入现有语言、图和工具链。模式是实时系统中的一个基本概念，也是设计者思考系统时的重要方式。它们超越了状态机中对底层有状态行为的描述。模态协调在此语境下，模态协调能够编排与不同运行模式相关的复杂事件处理网络。此外，借助模式局部时间，它为定时行为建模提供了强大工具。暂停和继续模式局部行为的能力，在 LF 中若要实现，通常会十分繁琐。

尽管模式已经成为众多现有程序设计语言和建模语言的核心，但将模态抽象构建到多语言协调语言 LF 中的方法具有一项优势：能够同时适用于一系列目标语言。程序员仍然可以使用 LF 所支持的任意目标语言开发底层“业务逻辑”，并且可以仅使用 LF 以精简、确定性的方式表达模态方面，同时获得图示支持，而基本无需考虑还会使用哪些其他 LF 功能。当前，我们的实现为 C语言 和 Python 目标提供了模态支持，展示了该方法的灵活性。尽管未来还将支持其他目标，C语言 目标体现了该方法对嵌入式底层应用的适用性，而 Python 则展示了其与一种常用于云和机器学习应用的高级脚本语言的兼容性。我们还成功地在 LF 中使用模式来控制一个机器人，并为驾驶和避障指定不同的模式。该方法将用于教授学生嵌入式系统的建模与设计。

模态反应器的一项关键设计决策是保留协调语言（LF）与目标语言（C语言、Python、Rust 等）之间清晰的分离。在 LF 层面，我们表达模态结构，包括封装组件的模式以及转换。我们还进一步定义哪些模式是初始模式，以及哪些反应可以触发哪些转换。不过，涉及调用转换和产生效果的实际运行时行为，则通过反应在目标语言中处理。然而，目标语言的作用仅限于触发转换，整个模态基础设施及其行为均由系统自动生成。由于 LF 层面存在这种分离，模式只是对实际行为的抽象。某个模式是否确实可达，并未在 LF 层面加以规定，而是取决于具体实现。

虽然模式的静态方面已提供一定的结构信息，可用于模型检查，但一个自然的问题是，能否且是否值得对两者进行整体分析。在一定程度上，编译期间已经开展了这类分析，例如检查类型兼容性，或者在节点的情况下，检查模式转换的一致性。然而，正如 VI-E 节所讨论的，未来可能的一项工作是以“白盒”方式对 LF 程序进行模型检查，同时考虑目标语言，例如 SPIN [14]。我们还计划开发用于验证和模型检查模态反应器的工具。另一个方向是探索能够协调联邦式 LF 程序的模态模型。

最后，我们有兴趣探索模式或其他（最小化的）语言扩展能否用于表达 LF 中的行为树。

参考文献

[1] S. Andalam、P. S. Roop 和 A. Girault，“使用 PRET-C 实现嵌入式应用的可预测多线程”，载于《协同设计的形式化方法与模型》（MEMOCODE）。IEEE/ACM，2010年，会议论文集，第159–168页。[2] C. Andr´e，“计算 SyncCharts 的反应”，《理论计算机科学电子笔记》，第88卷，第3–19页，2004年10月。[3] A. Benveniste、P. Caspi、S. Edwards、N. Halbwachs、P. Le Guernic 和 R. de Simone，“同步语言十二年后”，《IEEE会刊》，第91卷，第1期，第64–83页，2003年。[4] F. Boussinot，“Fairthreads：在 C 语言中混合协作式线程与抢占式线程”，《并发与计算：实践与经验》，第18卷，第5期，第445–469页，2006年4月。[5] J.-L. Colac¸o、G. Hamon 和 M. Pouzet，“在同步数据流系统中混合信号与模式”，载于ACM嵌入式软件国际会议（EMSOFT’06）。韩国首尔：ACM，2006年10月，第73–82页。[6] J. Colac¸o、B. Pagano 和 M. Pouzet，“SCADE 6：用于开发嵌入式关键软件的形式语言（特邀论文）”，载于第11届软件工程理论方面国际研讨会 TASE，法国索菲亚·安提波利斯，2017年9月，第1–11页。[7] M. E. Conway，“可分离转移图编译器的设计”，《ACM通讯》，第6卷，第7期，第396–408页，1963年。[8] F. Dabek、N. Zeldovich、F. Kaashoek、D. Mazi`eres 和 R. Morris，“面向健壮软件的事件驱动编程”，载于第10届ACM SIGOPS欧洲研讨会论文集，丛书 EW 10。美国纽约州纽约：计算机协会，2002年，第186–189页。[9] S. A. Edwards 和 J. Hui，“稀疏同步模型”，载于2020年规格与设计语言论坛（FDL），2020年，第1–8页。[10] K. Furuta、M. Yamakita 和 S. Kobayashi，“使用伪状态反馈控制倒立摆摆起”，《机械工程师学会会刊》，I辑：《系统与控制工程杂志》，第206卷，第4期，第263–269页，1992年。[11] G. Hamon 和 J. Rushby，“Stateflow 的操作语义”，载于软件工程基础方法国际会议。施普林格，2004年，第229–243页。[12] D. Harel，“Statecharts：复杂系统的可视化形式体系”，《计算机编程科学》，第8卷，第3期，第231–274页，1987年。[13] C. Hewitt，“将控制结构视为消息传递模式”，《人工智能杂志》，第8卷，第3期，第323–363页，1977年。[14] G. J. Holzmann，“使用 SPIN 对 ANSI-C 代码进行逻辑验证”，载于第7届SPIN模型检验与软件验证国际研讨会论文集。柏林、海德堡：施普林格出版社，2000年，第131–147页。[15] L. Lavagno 和 E. Sentovich，“ECL：系统级设计的规格说明环境”，载于第36届ACM/IEEE设计自动化会议论文集（DAC’99）。ACM，1999年，第511–516页。[16] E. A. Lee，“线程的问题”，《计算机》，第39卷，第5期，第33–42页，2006年。[17] E. A. Lee 和 S. Tripakis，“Ptolemy 中的模态模型”，载于第3届基于方程的面向对象建模语言与工具国际研讨会（EOOLT），第47卷。林雪平大学电子出版社，林雪平大学，2010年，第11–21页。[18] E. A. Lee、H. Zheng 和 Y. Zhou，“因果接口与组合因果分析”，《接口技术基础》（FIT），CONCUR卫星会议，旧金山，加利福尼亚州，第2卷，第402–405页，2005年。[19] J. Liu、J. Eker、J. W. Janneck 和 E. A. Lee，“嵌入式控制系统的逼真仿真”，《IFAC会议论文集》，第35卷，第1期，第391–396页，2002年。[20] M. Lohstroh，“反应器：反应式系统并发计算的确定性模型”，博士论文，电气工程与计算机科学系，加州大学伯克利分校，2020年12月。 [在线]。网址：http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html [21] M. Lohstroh、´I. ´Incer Romeo、A. Goens、P. Derler、J. Castrillon、E. A. Lee 和 A. Sangiovanni-Vincentelli，“反应器：可组合反应式系统的确定性模型”，载于第8届基于模型的网络物理系统设计国际研讨会（CyPhy’19），第LNCS 11971卷。施普林格出版社，2019年，会议论文集，第27页。[22] M. Lohstroh、C. Menard、S. Bateni 和 E. A. Lee，“迈向确定性并发系统的通用语言”，《ACM嵌入式计算系统汇刊》（TECS），FDL’19特刊，第20卷，第4期，文章36，2021年5月。

[23] M. Lohstroh、C. Menard、A. Schulz-Rosengarten、M. Weber、J. Castrillon 和 E. A. Lee，“一种用于跨多个时间线进行确定性协调的语言”，载于 *2020*年规格与设计语言论坛 *(FDL)*。IEEE，2020 年，第 1–8 页。[24] M. C. Loring、M. Marron 和 D. Leijen，“异步 JavaScript 的语义”，*SIGPLAN* 通讯，第 52 卷，第 11 期，第 51–62 页，2017 年 10 月。[25] Z. Manna 和 A. Pnueli，“验证混合系统”，载于 混合系统。施普林格，1992 年，第 4–35 页。[26] R. Roestenburg、R. Bakker 和 R. Williams，《*Akka*实战》。Manning Publications Co.，2016 年。[27] M. Samek，*C/C* 中的实用 *UML* 状态图++面向嵌入式系统的事件驱动程序设计。Newnes，2008 年。[28] A. Schulz-Rosengarten、R. von Hanxleden、F. Mallet、R. de Simone 和 J. Deantoni，“SCCharts 中的时间”，载于 规格与设计语言论坛论文集 *(FDL ’18)*，德国慕尼黑，2018 年 9 月。[29] C. D. Schulze、M. Sp¨onemann 和 R. von Hanxleden，“带端口约束的分层图绘制”，《可视化语言与计算期刊》，图示美学与布局特刊，第 25 卷，第 2 期，第 89–106 页，2014 年。[30] B. Stroustrup，“什么是面向对象编程？”《*IEEE*软件》，第 5 卷，第 3 期，第 10–20 页，1988 年。[31] R. von Hanxleden，“C 中的 SyncCharts——一种轻量级确定性并发方案”，载于 *ACM*嵌入式软件会议 *(EMSOFT)*，2009 年，会议论文集，第 11–16 页。[32] R. von Hanxleden 等，“SCCharts：面向安全关键应用的顺序构造式状态图”，载于 *ACM SIGPLAN* 编程语言设计与实现会议，丛书 PLDI ’14。美国纽约：ACM，2014 年，第 372–383 页。[33] R. von Hanxleden、E. A. Lee、H. Fuhrmann、A. Schulz-Rosengarten、S. Domr¨os、M. Lohstroh、S. Bateni 和 C. Menard，“十二年后的语用学：Lingua Franca 报告”，载于 第 *11* 届形式化方法、验证与确认应用国际研讨会 *(ISoLA)*，《计算机科学讲义》丛书，第 13702 卷。希腊罗得岛：施普林格，2022 年 10 月，第 60–89 页。[34] F. Wagner、R. Schmuki、P. Wolstenholme 和 T. W. Thomas，使用有限状态机建模软件：实用方法。Auerbach Publications，2006 年。[35] E. Yip、A. Girault、P. S. Roop 和 M. Biglari-Abhari，“面向多核处理器的 ForeC 同步确定性并行编程语言”，载于 第 *10* 届 *IEEE* 嵌入式多核/众核片上系统国际研讨会，*MCSOC 2016*，法国里昂，*2016* 年 *9* 月 *21–23* 日。IEEE 计算机学会，2016 年，第 297–304 页。