

有限状态机FSM

反应式系统

- 反应式(reactive)系统就是指能够持续地与环境进行交互，并且及时地进行响应
 - 例如视频监控系统会持续监测，并当有陌生人闯入时立刻触发警报
- 大多数实时系统都是反应式系统，如通信网络、看门狗和家庭应用（洗衣机、微波炉和洗碗机等）等

特征

- 其特征是事件驱动，当系统接收到一个外部事件时，需要对事件进行处理，然后产生响应。反应式系统可以定义为系统可能的输入/输出事件序列的集合、条件、动作和时序约束。
- 对反应式系统建模的主要模型包括有限状态机、行为有限状态机、协同设计有限状态机、UML状态图（UML StateCharts）、程序状态机和通信交互进程等。

FSM定义

- 有限状态机 (finite-state machine, FSM) 又称有限状态自动机 (finite-state automaton, FSA) , 简称状态机, 是表示有限个状态以及在这些状态之间的转移和动作等行为的数学计算模型 (Wikipedia)
- 状态存储关于过去的信息, 就是说: 它反映从系统开始到现在时刻的输入变化。
- 转移指示状态变更, 并且用必须满足确使转移发生的条件来描述它。
- 动作是在给定时刻要进行的活动的描述。有多种类型的动作:
 - 进入动作 (entry action) : 在进入状态时进行
 - 退出动作 (exit action) : 在退出状态时进行
 - 输入动作: 依赖于当前状态和输入条件进行
 - 转移动作: 在进行特定转移时进行

Finite State Machines

- $\text{FSM} = ($
 - {Input symbols},
 - {Output Symbols},
 - {States},
 - {Initial States},
 - Transition Relation (mapping of Input Symbol, Current State to next State(s))
 - Output Function (mapping of Input Symbol, Current State to Current Output Symbol) $)$
-

应用

- 除了建模这里介绍的反应系统之外，有限状态自动机在很多不同领域中是重要的，包括电子工程、语言学、计算机科学、哲学、生物学、数学和逻辑学。
- 有限状态机是在自动机理论和计算理论中研究的一类自动机。在计算机科学中，有限状态机被广泛用于建模应用行为、硬件电路系统设计、软件工程，编译器、网络协议等研究。

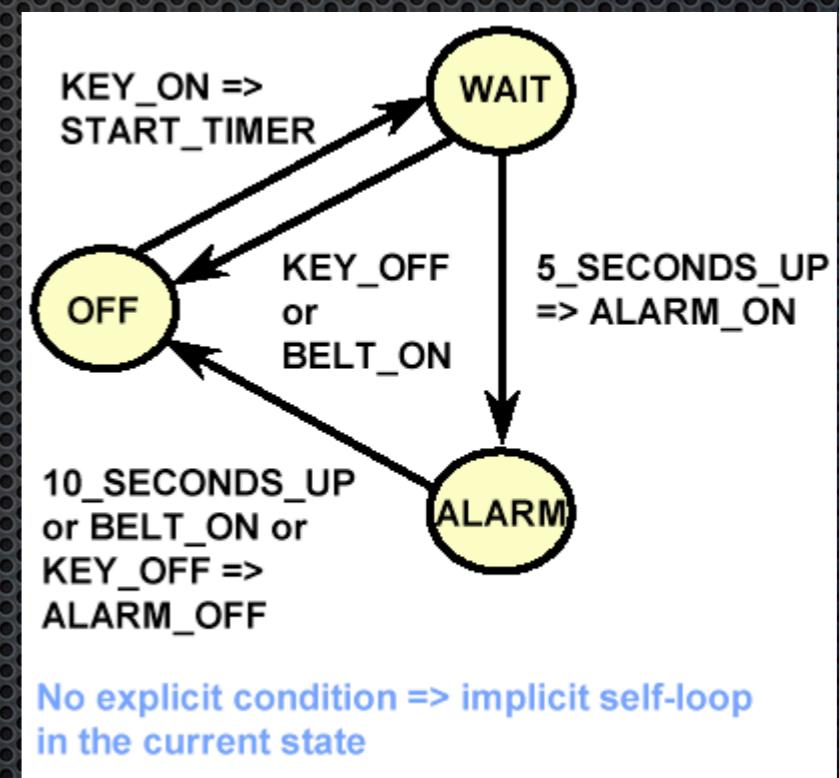
FSM应用实例

- 场景

- 如果驾驶员用钥匙启动汽车，在5秒内没有系好安全带，则警报器发出5秒警报，直到驾驶员系好安全带或者关闭发动机

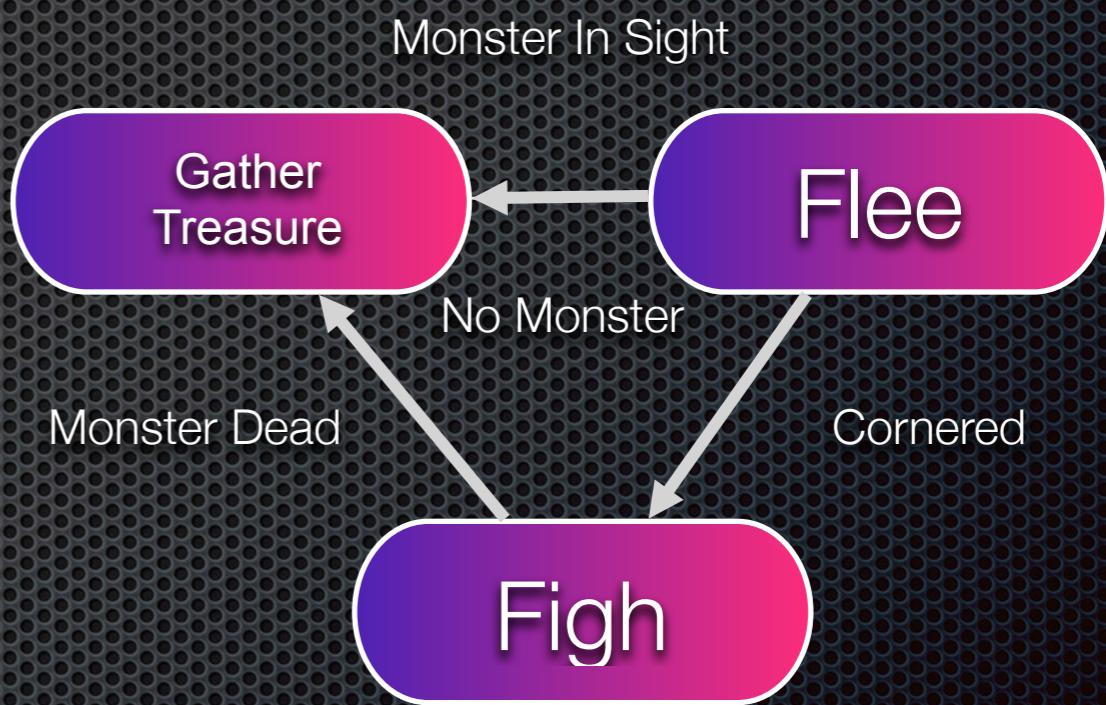
- 形式化规格

- Inputs = {KEY_ON, KEY_OFF, BELT_ON, BELT_OFF, 5_SECONDS_UP, 10_SECONDS_UP}
- Outputs = {START_TIMER, ALARM_ON, ALARM_OFF}
- States = {Off, Wait, Alarm}
- Initial State = off
- NextState: CurrentState, Inputs -> NextState
 - e.g. NextState(WAIT, {KEY_OFF}) = OFF
- Outs: CurrentState, Inputs -> Outputs
 - e.g. Outs(OFF, {KEY_ON}) = START_TIMER



游戏中的FSM

- 角色AI可以建模为一系列认知状态
- 环境事件可以迫使状态发生迁移
- 即使对于非程序员来说，认知模型也很容易掌握。

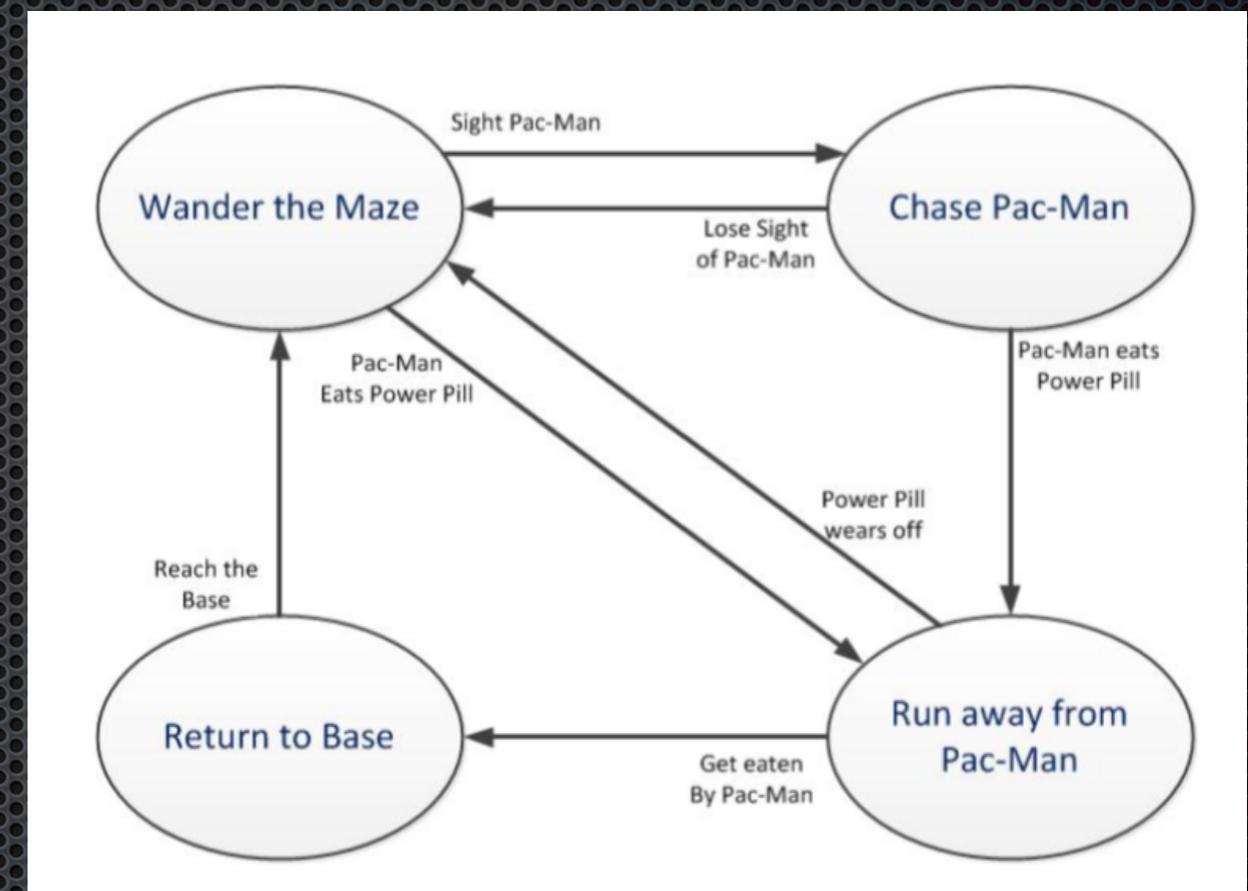


Pac-Man

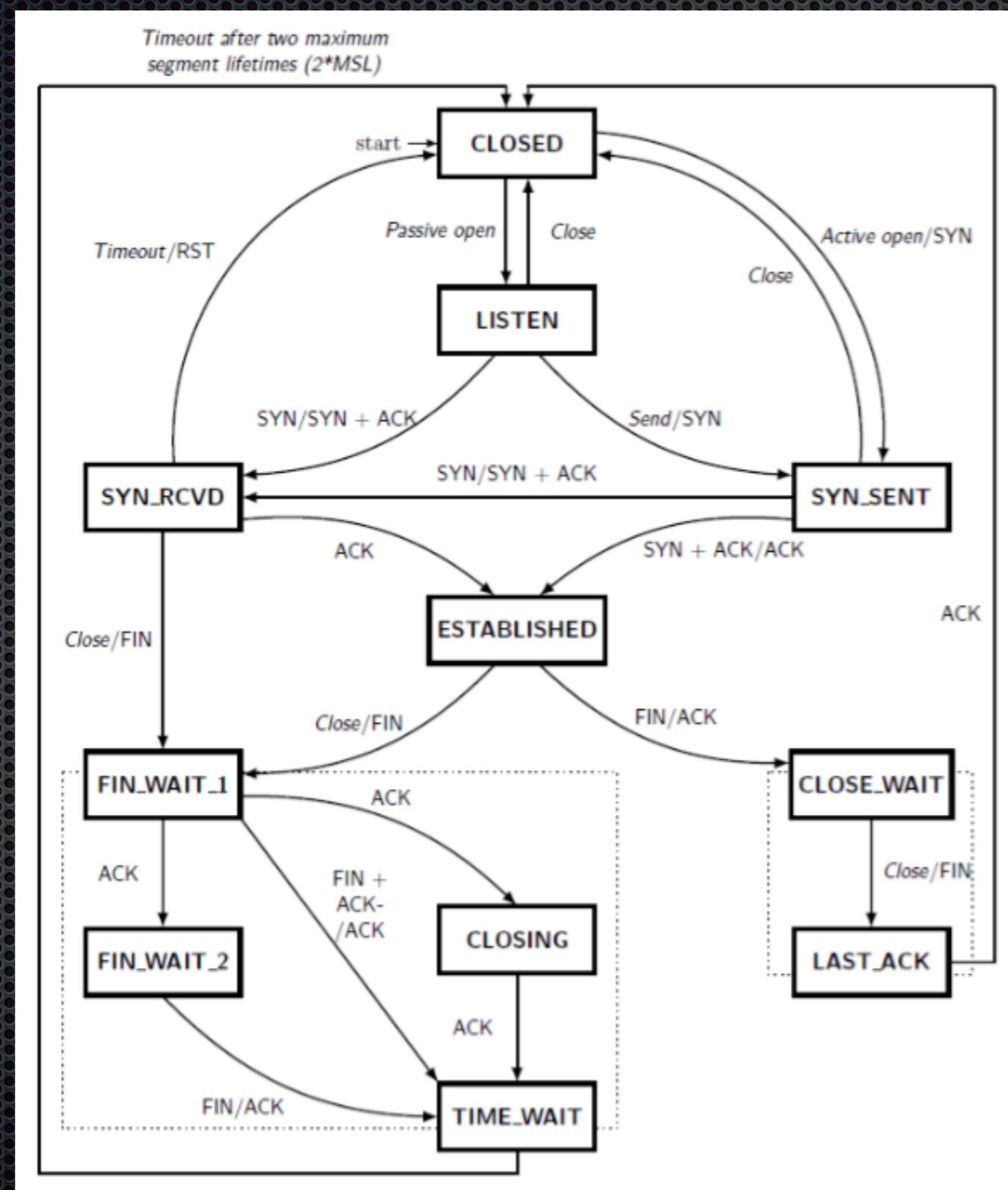


Pac-man

- 吃豆人中的幽灵有四种行为：
 - 在迷宫中随机漫步
 - 当吃豆人在视线范围内追逐吃豆人
 - 当吃豆人吃了一个能量球时，逃离吃豆人
- 回到中央基地再生



Internet Protocols: TCP



FSM特性

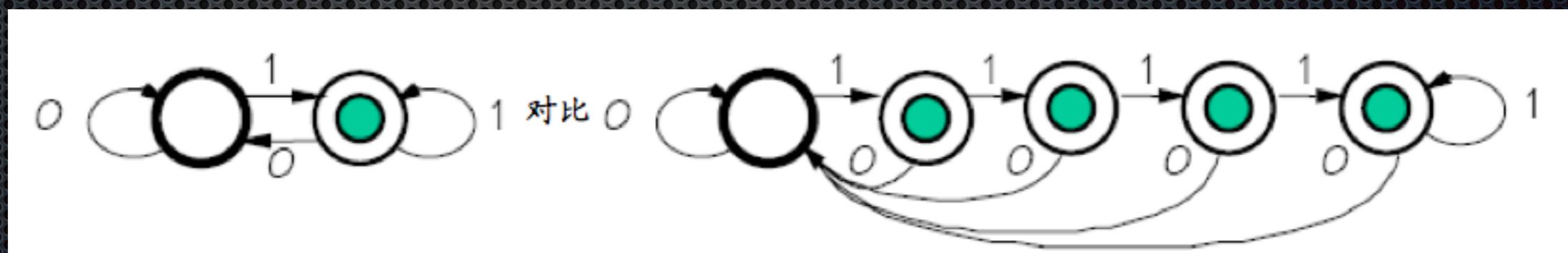
- $(\text{next_states}, \text{output}) = \Phi(\text{state}, \text{event})$
- 有限状态机具有两个重要特性
 - 确定性：如果对于每个状态，每个输入值最多可激活一个转移，则称这样的状态机具有确定性，这意味着 $\Phi(S, E)$ 是单一值
 - 可接受性：如果对于每个状态，每个输入都有至少一个可能的转移，则称这样的状态机为可接受的，定义了每个可能的状态和输入值
-

Moore机和Mealy机

- 状态模型是单线程的
 - 任何时候只有一个状态是有效的
- Moore状态模型：意味着输出完全由当前状态决定，与输入信号的当前值无关
 - 非反应性，输入对输出的影响要到下一个时钟周期才能反映出来
 - 结构简单，状态数量大于等于对应的mealy机中的数量
- Mealy状态模型：意味着输出既依赖于当前状态，也与输入信号的当前值有关
 - 往往更加精简
 - 即时对输入产生输出，即响应速度快
- 可相互转换，mealy机也可转换成大致等效的Moore机，差别只是输出产生于下一个响应，而非当前响应

等价与优化

- 对所有的输入具有相同输出的两个FSM被称为等价
- 等价不需要同构(相同形状)——也就是说，两个等价的FSM不需要有相同的图，甚至不需要有相同数量的状态!
- 等价使优化成为可能：优化一个FSM意味着缩减状态机的状态数目，同时保证状态机能实现同样功能

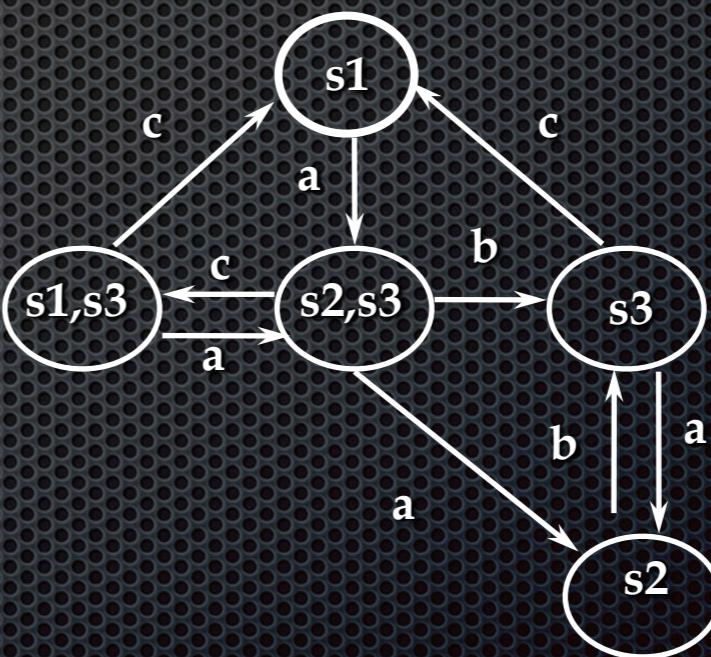
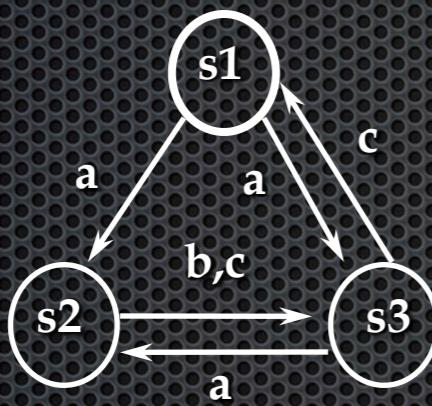


非确定性FSM

- 当下一个状态或输出有多个值时，FSM被称为非确定性的
 - 五元组中，转移关系表示状态和输入值映射到可能的（下一状态，输出值）对集，初始状态不是一个元素而是一个集合，可以有多个初始状态
- 在嵌入式系统建模中的主要用途：
 - 环境建模：对于隐藏与环境如何运作无关的细节非常有用
 - 机器人不确定的行为

DFA与NFA

- 形式上，确定有限状态自动机（DFA, Deterministic Finite Automaton）和非确定有限状态自动机（NFA）是等价的（幂集构造）
- 在实践中，NFA通常更简洁
-



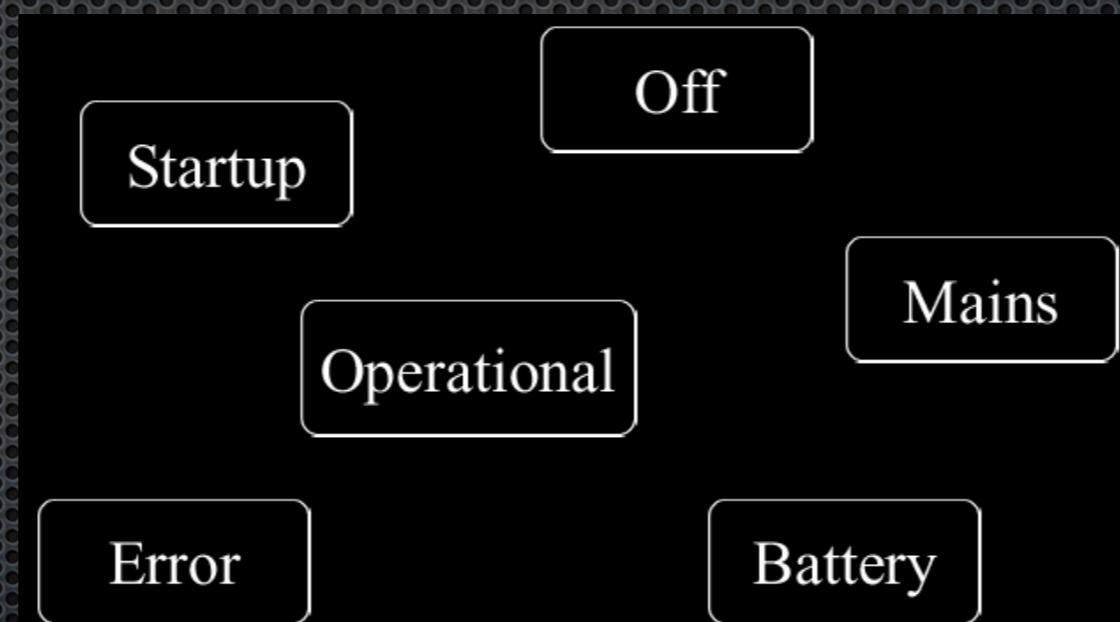
常规FSM建模

- 经常过度指定
 - 完全指定的(即使不关心)
- 由于缺乏组合潜力，可伸缩性较差
 - 状态的数量可能难以管理
- 不支持并发
 - 经常想要推理组合状态机的局部子属性，但是如何关联子状态的行为？
- 简单的解决方案：在模型中引入层次结构
 - 并不像听起来那么简单

并发

- 例子：

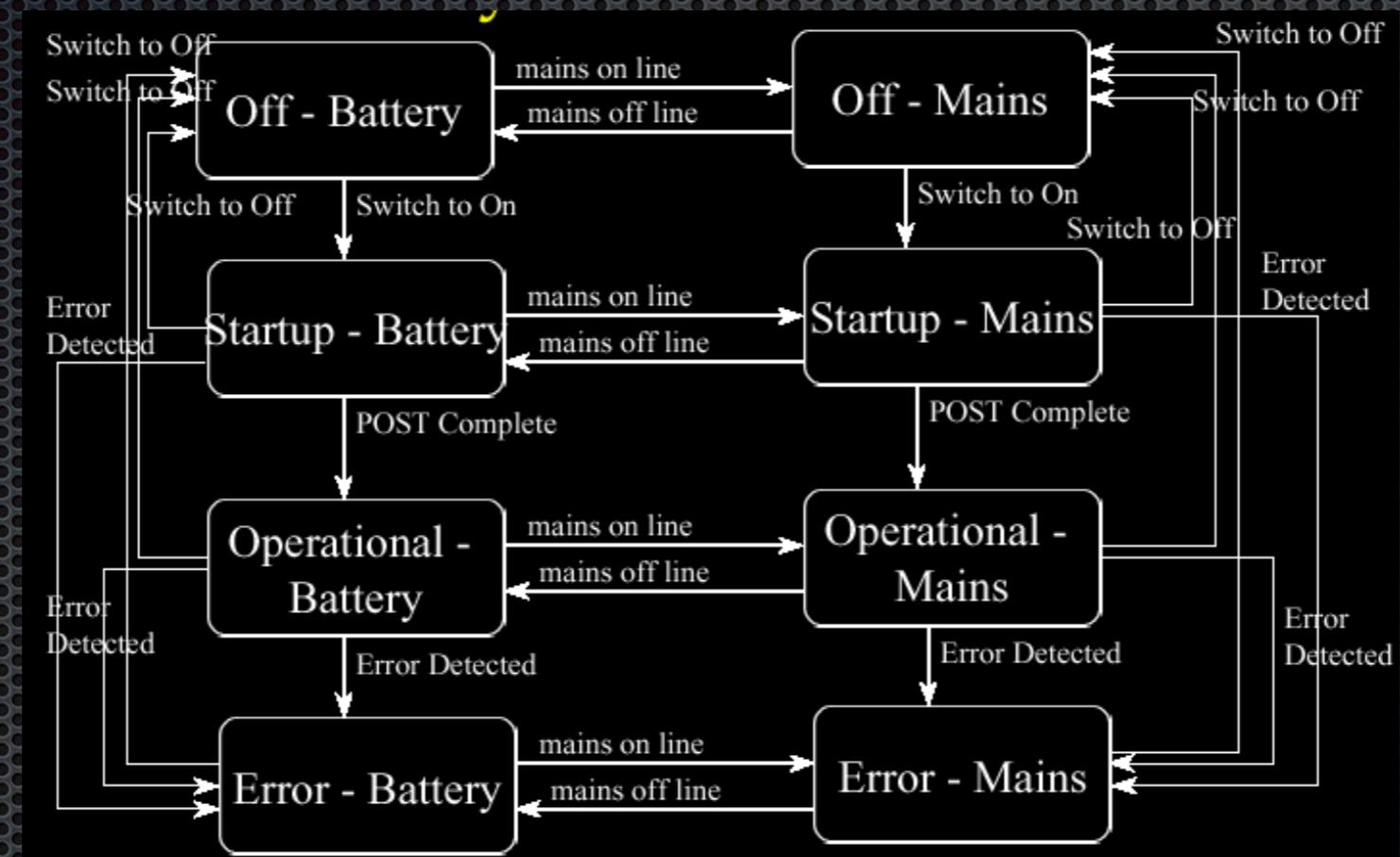
- 设备可以处于状态{关闭，启动中，运行中，错误}
- 从{电源，电池}运行时
- 如何安排这些状态？



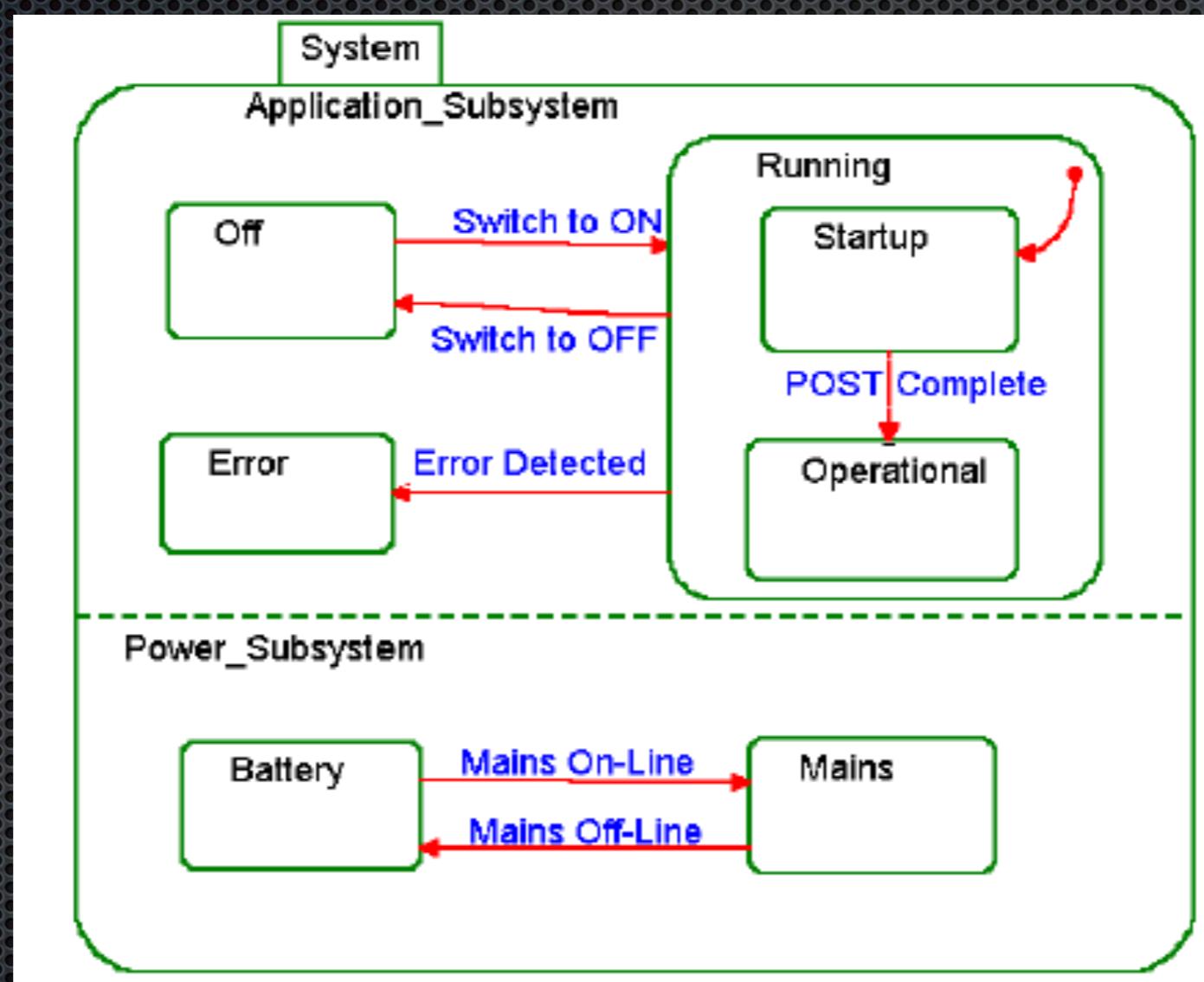
并发

- Mealy/Moore视角下的不同状态:
 - 电池供电时的操作
 - 电源供电时的操作
- 导致状态爆炸
- 解决方案?
 - 允许状态并发操作(拥抱NFA!)

Mealy-Moore解决方案



状态图



正交组件

myInstance: myClass	
tColor	Color
boolean	ErrorStatus
tMode	Mode

```
enum tColor {eRed, eBlue,  
eGreen};
```

```
enum boolean {TRUE,  
FALSE}
```

```
enum tMode {eNormal,  
eStartup, eDemo}
```

如何描述这些状态？

方法1：枚举所有状态

eRed, FALSE,
eDemo

eBlue, FALSE,
eDemo

eGreen, FALSE,
eDemo

eRed, TRUE,
eDemo

eBlue, TRUE,
eDemo

eGreen, TRUE,
eDemo

eRed, FALSE,
eNormal

eBlue, FALSE,
eNormal

eGreen, FALSE,
eNormal

eRed, TRUE,
eNormal

eBlue, TRUE,
eNormal

eGreen, TRUE,
eNormal

eRed, FALSE,
eStartup

eBlue, FALSE,
eStartup

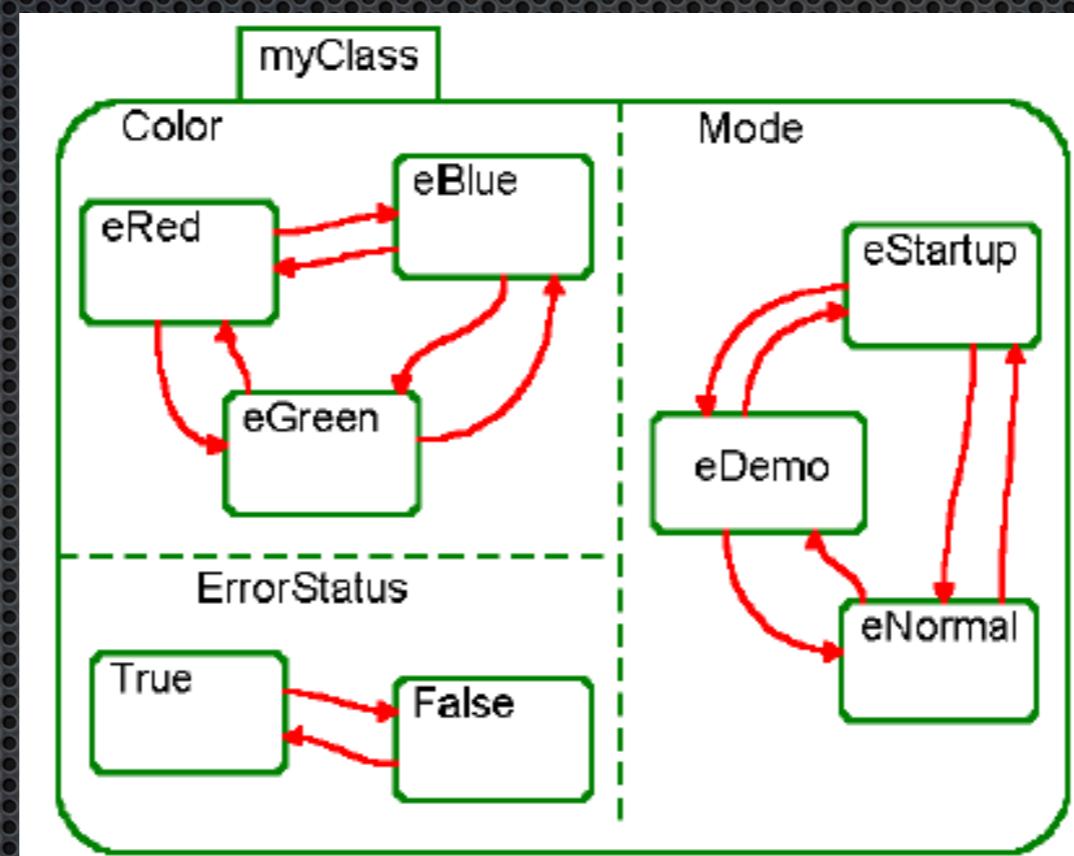
eGreen, FALSE,
eStartup

eRed, TRUE,
eStartup

eBlue, TRUE,
eStartup

eGreen, TRUE,
eStartup

分层(和组合)模型

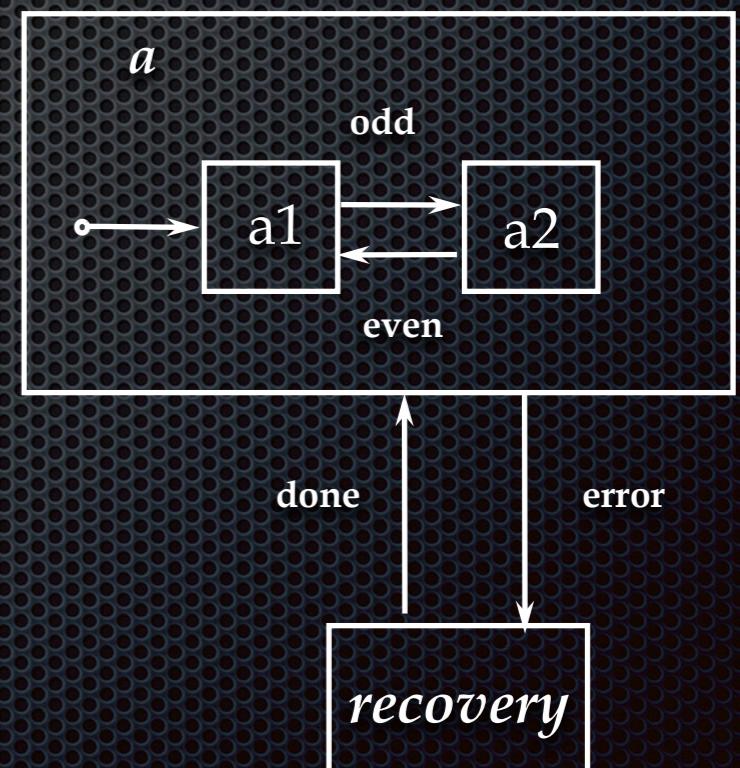


状态图: 层次FSM

- StateCharts语言由David Harel于1987年提出
- 支持层次结构模型及并发
- 状态图支持:
 - 将状态重复分解为AND/OR子状态
 - 嵌套状态、并发性、正交组件
 - 动作(可能有参数)
 - 活动(只要状态处于活动状态就执行的函数)
 - 判定式
 - 历史
 - 同步(即时广播)通信机制

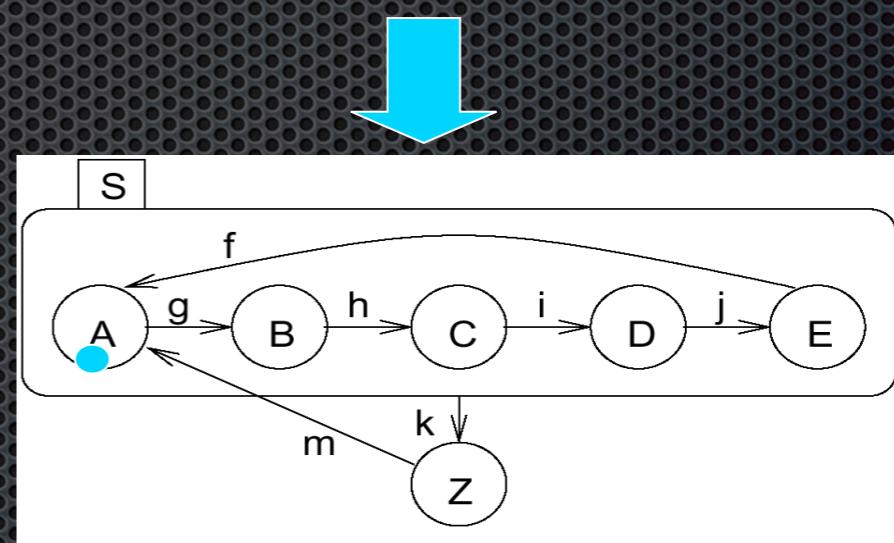
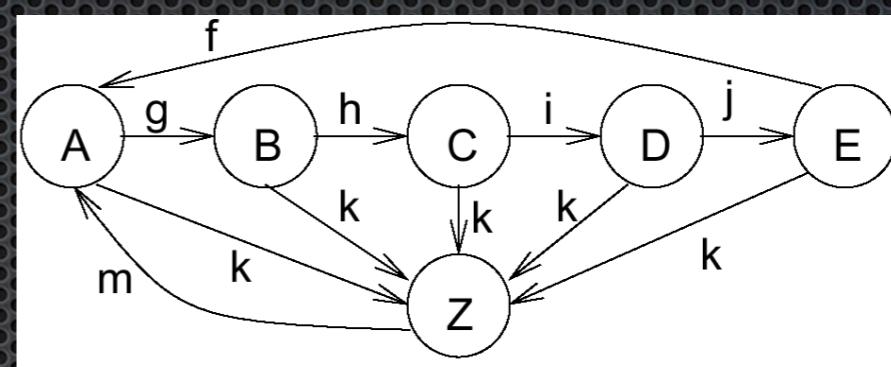
层次FSM模型

- 扩展有限状态机
- 层次结构:
 - 状态a “包含” 一个FSM
 - 处于a状态意味着FSM处于a状态
 - a的状态称为OR状态
 - 用于对抢占和例外进行建模
- 并发性:
 - 两个或两个以上FSM同时处于活动状态
 - 状态称为AND状态



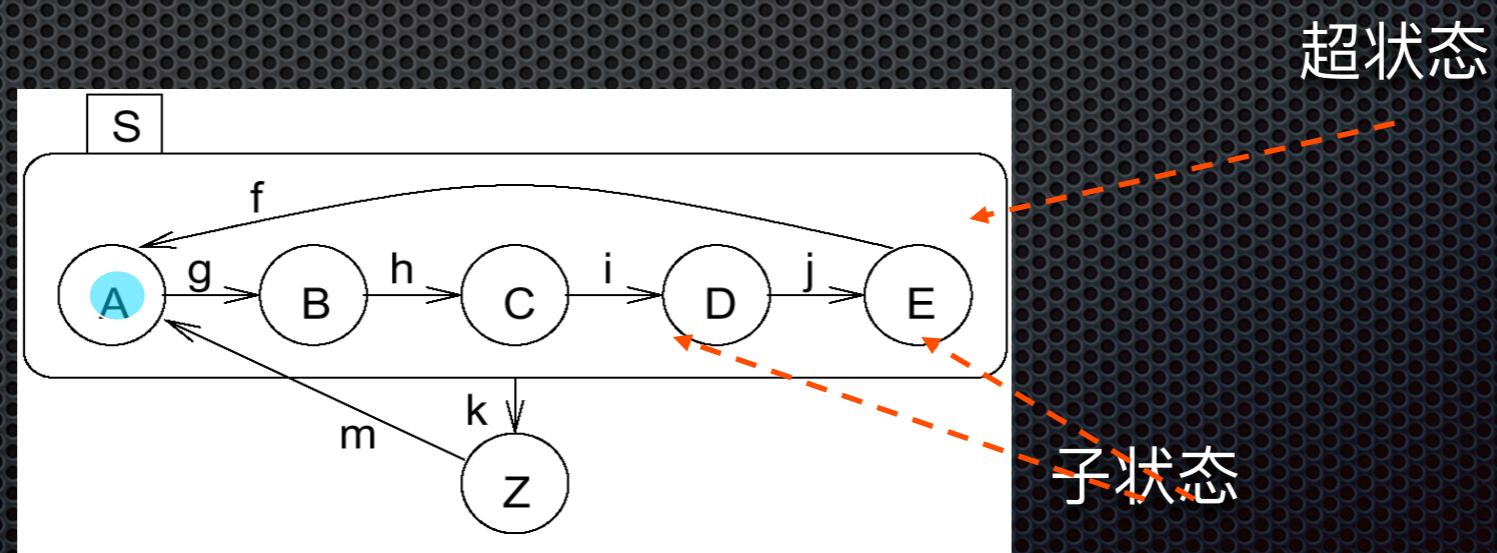
层次化状态图

- FSM将处于S的子状态之一
- 支持从/到子状态的转移
 - 初始状态(默认)
 - 历史



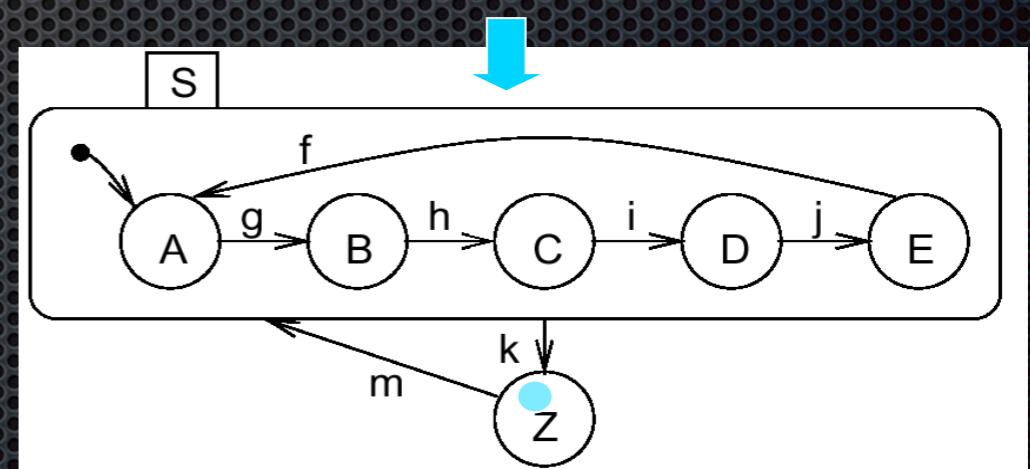
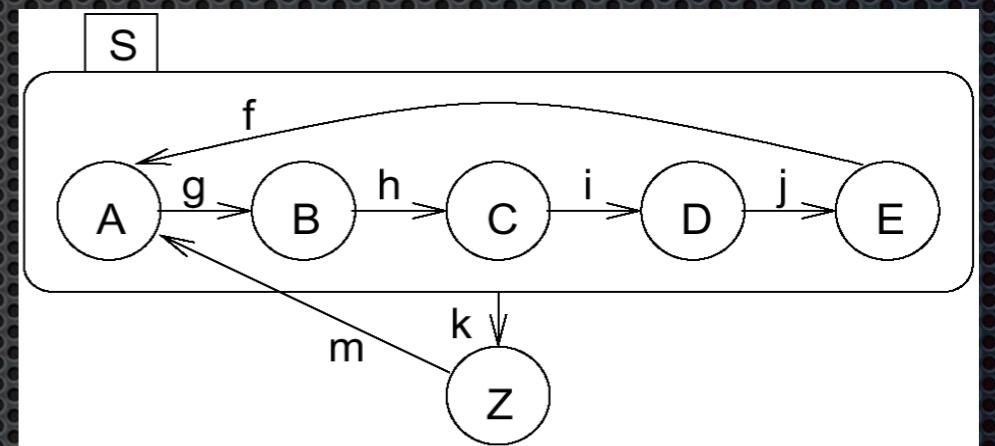
定义

- FSM的当前状态也称为活跃状态
- 包含其他状态的状态被称为超状态
- 包含在超状态中的状态被称为超状态的子状态
- 每个不包含其他状态的状态称为基本状态
- 对于每个基本状态S，包含S的超状态被称为祖先状态。
- 任何时候，下图所示FSM只能处于超状态S的某个状态中，则该类超状态被称为或型（OR）超状态



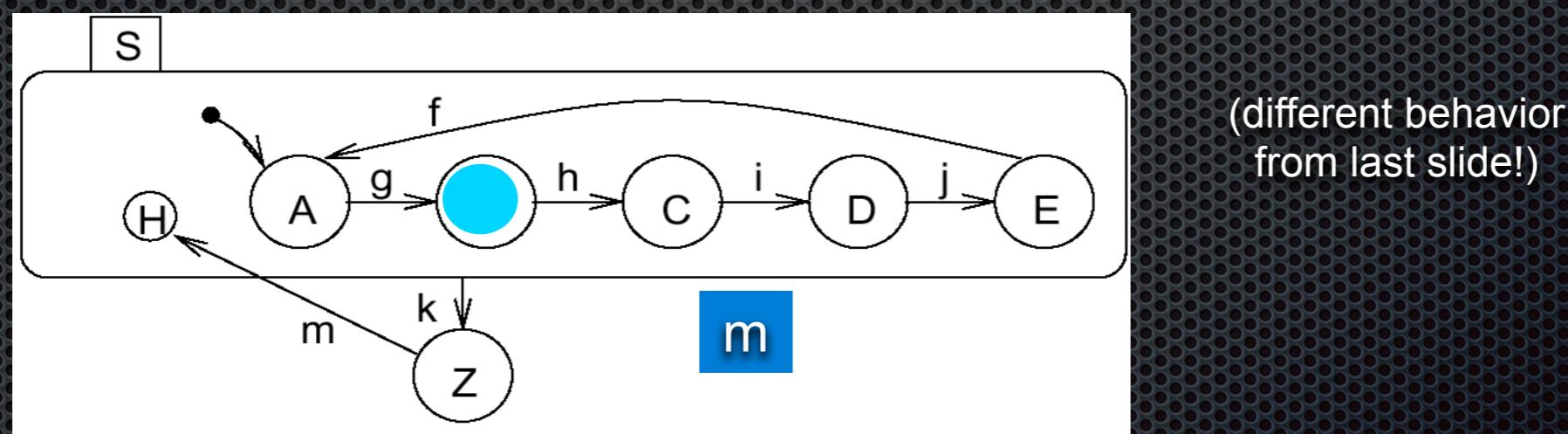
默认状态机制

- 指定超状态变为活跃时将要激活的特定子状态
 - 伪状态，不是一个状态

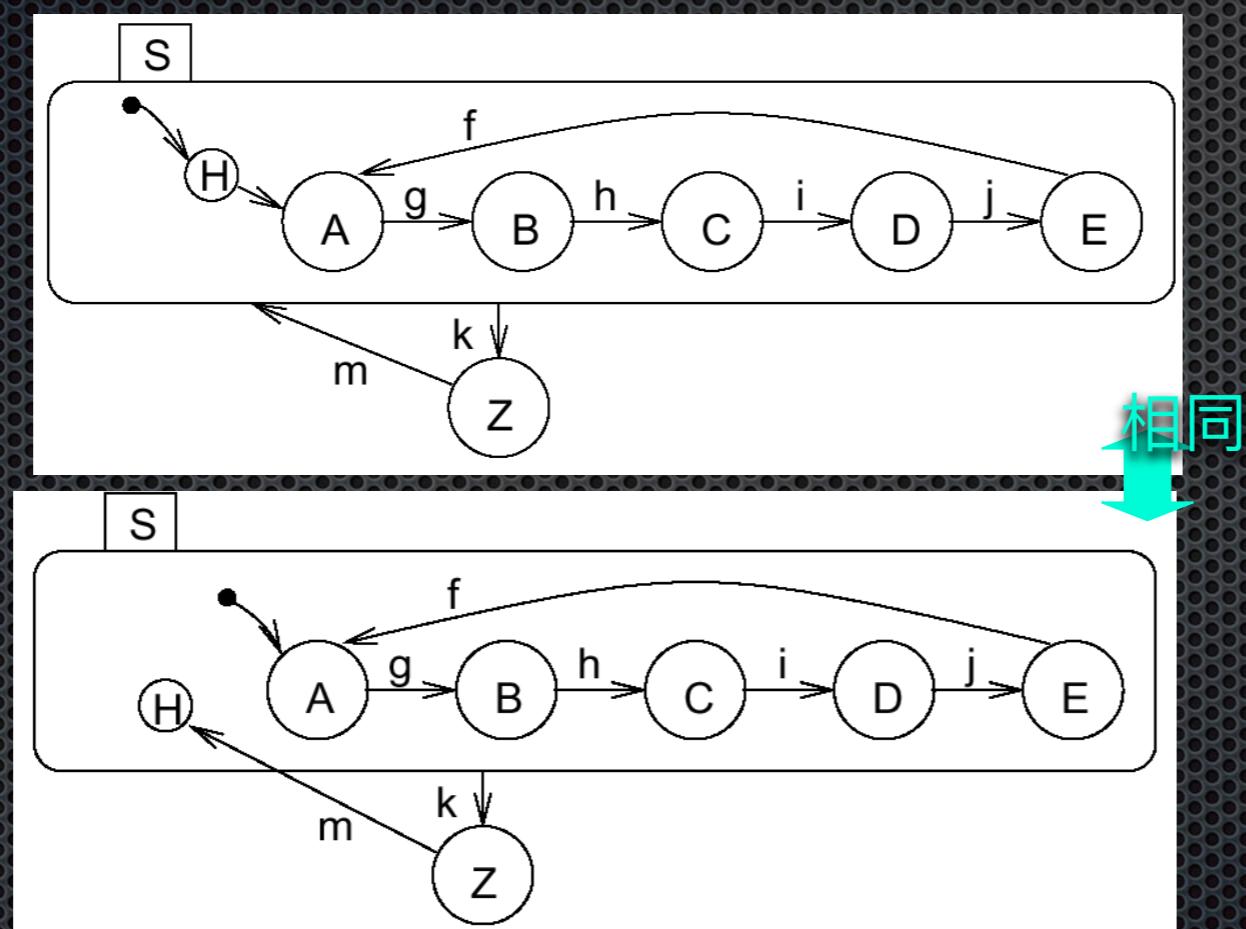


历史机制

- 采用这个机制，就可能回到超状态退出之前最后一个活跃子状态
 - 给定输入m, S返回到S离开之前的状态(可以是A、B、C、D或E)
- 在第一次进入S时，应用默认机制
- 历史和默认机制可以组合使用

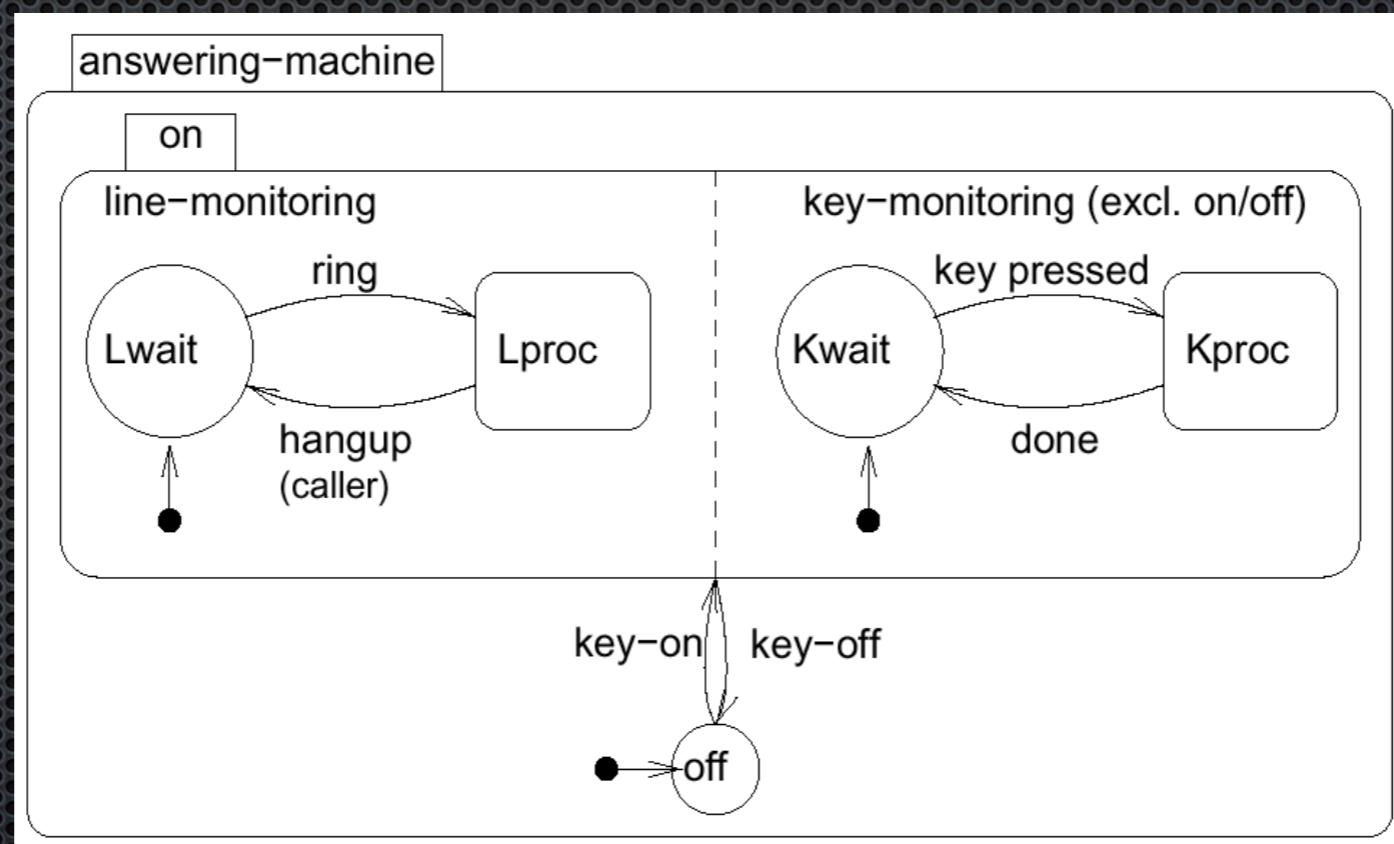


结合历史和默认状态机制



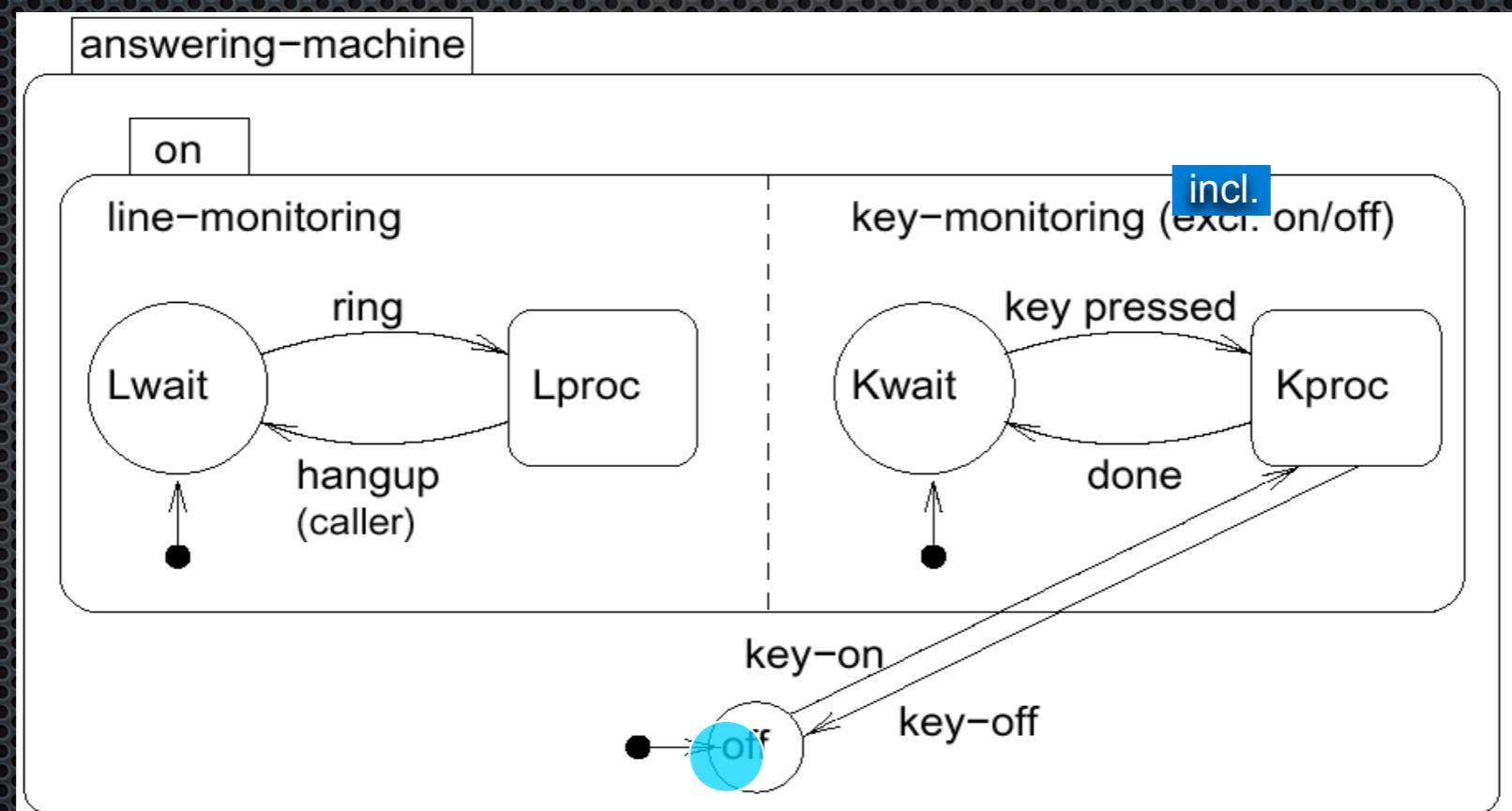
并发

- 与型超状态 (AND-super-state)：无论什么时候包含状态S的系统在进入S状态时都将进入S的所有子状态中
 - FSM处于超状态的所有(直接)子状态
 - 与或型超状态不同，需要多个控制点



进入和离开与型超状态

- 进入与型超状态时所进入的子状态可以被单独定义，可以是历史、默以及显示转移的任意组合

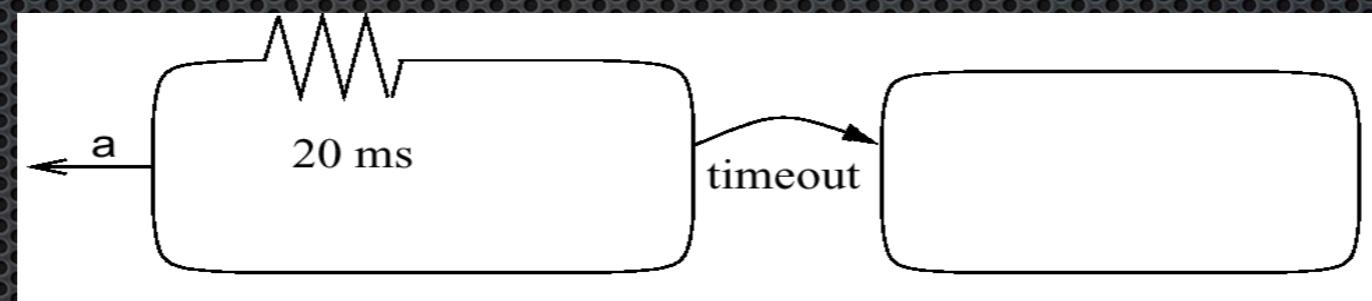


StateCharts图中的状态

- 与型状态
- 或型状态
- 基本状态

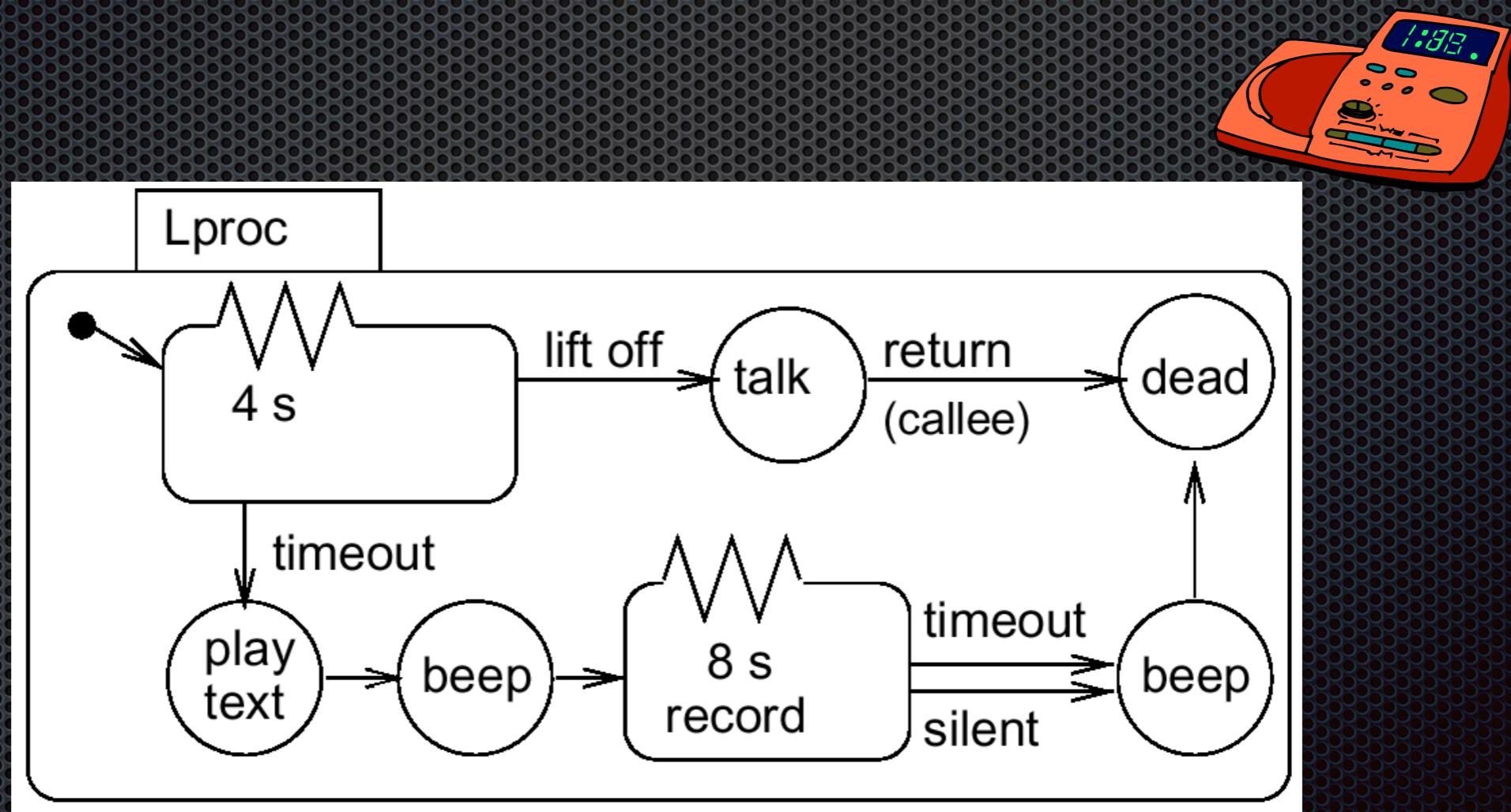
定时器

- 鉴于嵌入式系统对时间进行建模的需求，StateCharts提供了定时器
- 进入包含定时器的状态一段时间后，超时（timeout）将会发生，且系统将离开这个指定状态
- 定时器可以被分层使用



如果事件a在系统处于定时器状态的20毫秒内没有发生，则会发生超时

应答机层次结构中使用定时器



StateCharts中的边的标号

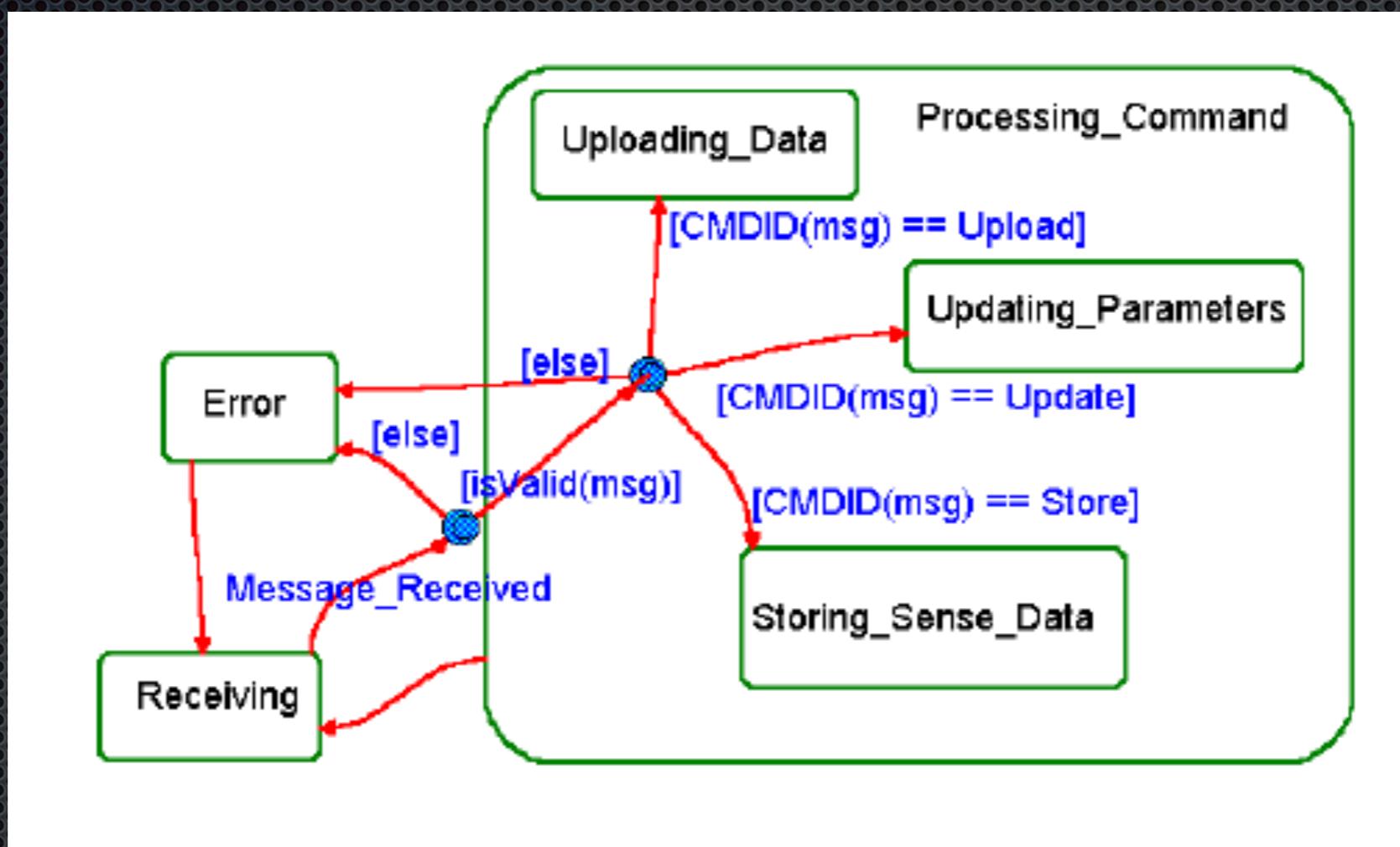
- 生成的输出可用边的标号来指定
- 在StateChart中标记转移的表达式的一般语法是“事件[条件]/动作”
(event[condition]/action)，其中
 - event是指触发转移的事件
 - condition部分隐含了变量值的测试或对系统当前状态的测试
 - action部分描述FSM对状态转移的反应
- 对于每个转移，事件、条件和动作都是可选的



示例

- on-key/on:=1(事件测试及变量赋值)
- [on=1] (对变量值进行条件测试)
- off-key[not in Lproc]/on:=0 (事件测试、对状态的条件测试、变量赋值，当该事件发生、条件为真时，执行赋值操作)

条件转移



StateCharts的优势

- 层次结构允许任意嵌套与型和或型超状态
- StateMate的语义定义在足够详细的层次上
- 大量的商业仿真工具可用
 - StateMate, StateFlow, BetterState, ...
- 可在后端将StateCharts转换为C或VHDL，从而支持软件或硬件实现

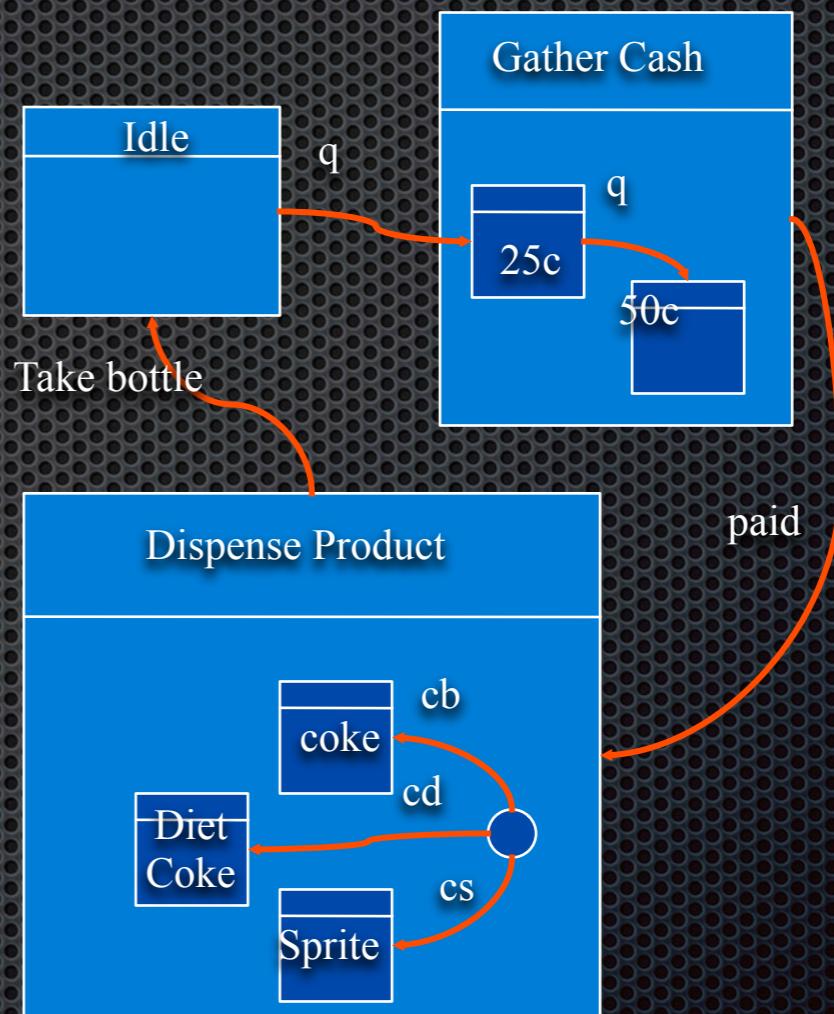
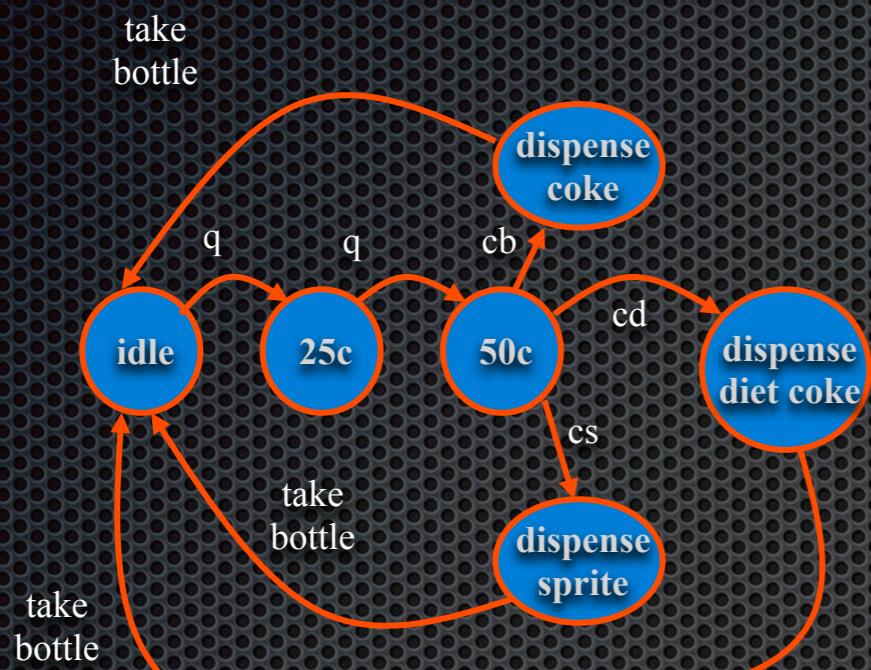
StateChart的不足

- 生成的C程序可能效率低下
- 生成的硬件可能更糟
- 难以应用于分布式应用程序
- 没有结构化层次的描述

例子：自动饮料售卖机

- 假设有一台自动饮料售卖机：
 - 开机后，机器等待投币
 - 当投入一个25分硬币时，机器等待另一个25分硬币
 - 当第二枚硬币存入时，机器进入等待状态
 - 当使用者按下“可乐”键时，就会出现可乐
 - 当使用者拿起瓶子时，机器再次进入等待状态
 - 当用户按下“雪碧”或“健怡可乐”时，就会出现雪碧或健怡可乐
 - 当使用者拿起瓶子时，机器再次进入等待状态

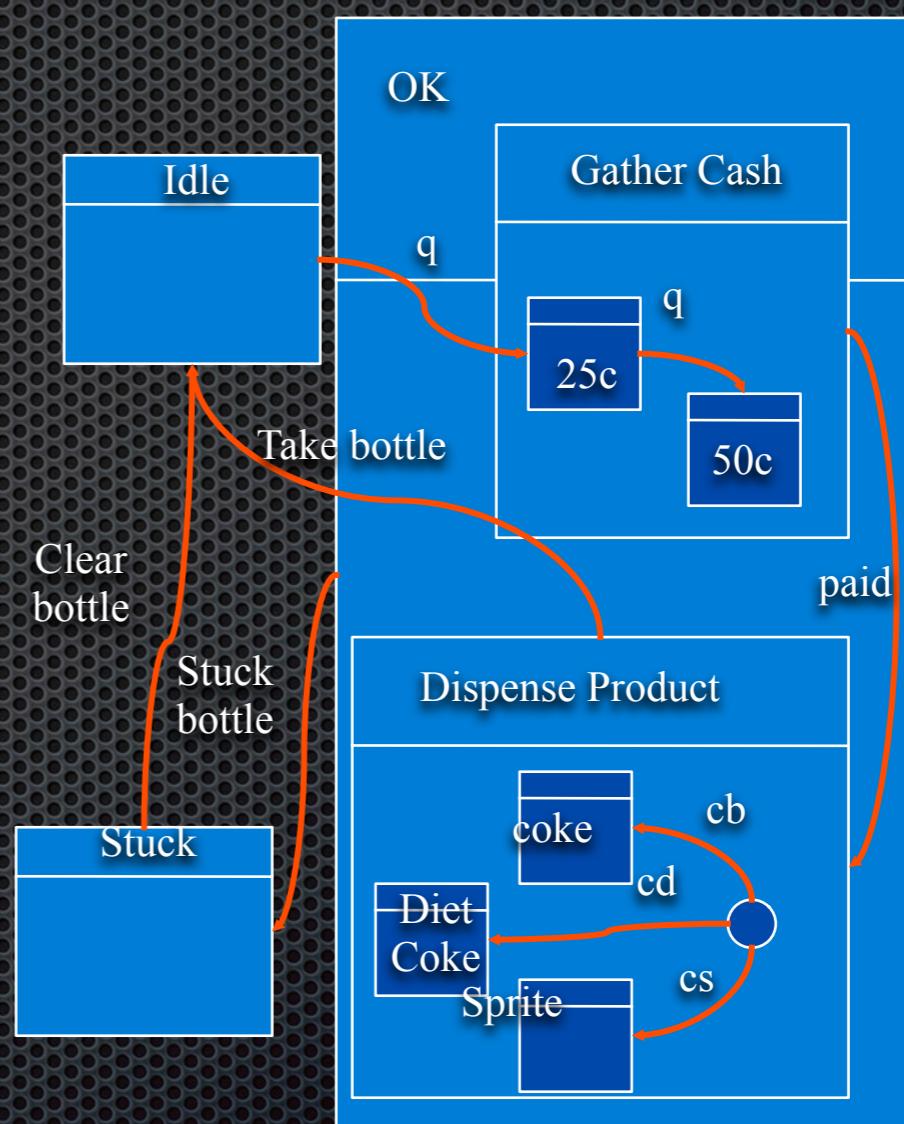
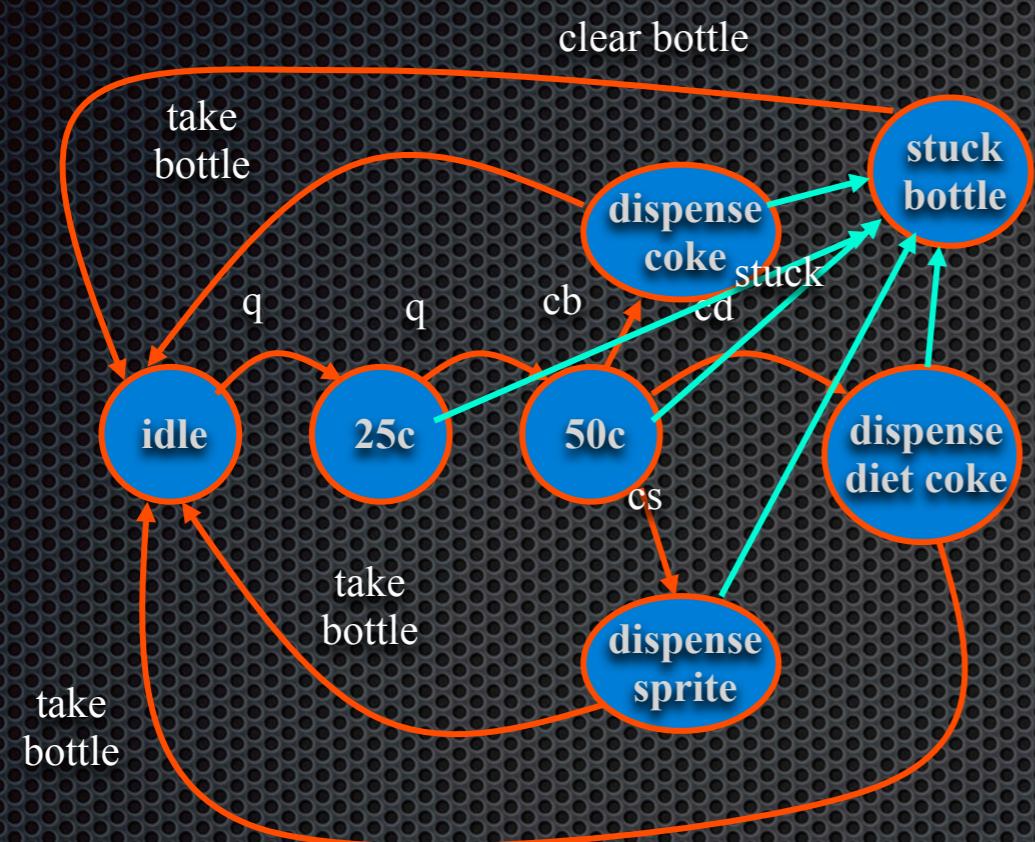
自动饮料售卖机 1.0



自动饮料售卖机V1.1

- 瓶子会在机器里卡住
 - 当瓶子卡住时，指示器会自动通知系统
 - 当这种情况发生时，机器将不再接受任何钱币或释放任何饮料瓶，直到饮料瓶被清理干净
 - 当被卡住的饮料瓶被清空时，机器将再次进入等待状态
- 状态机需要修改
 - 需要多少个新状态？
 - 需要多少新的转移？

自动饮料售卖机V1.1



层次FSM

- 层次结构允许
 - 合理的默认活动
 - 在多个地方可以增强行为
 - 考虑被卡住的恢复行为
 - 所需的状态数量大幅减少
 - 易于添加扩展状态语义
 - 增强状态和条件动作
- 我们如何在软件中进行有效的实现?
 - 层次FSM比FSM更复杂，成本更高
 - 但是要在表现力与复杂性和可维护性之间进行权衡

参考

- 嵌入式系统导论：CPS方法
- 嵌入式系统设计：CPS与物联网应用
- Practical UML STATECHARTS in C/C++, Second Edition.
Chapter 3