State Machines and State Charts

Part 1

Introduction

Bruce Powel Douglass, Ph.D.

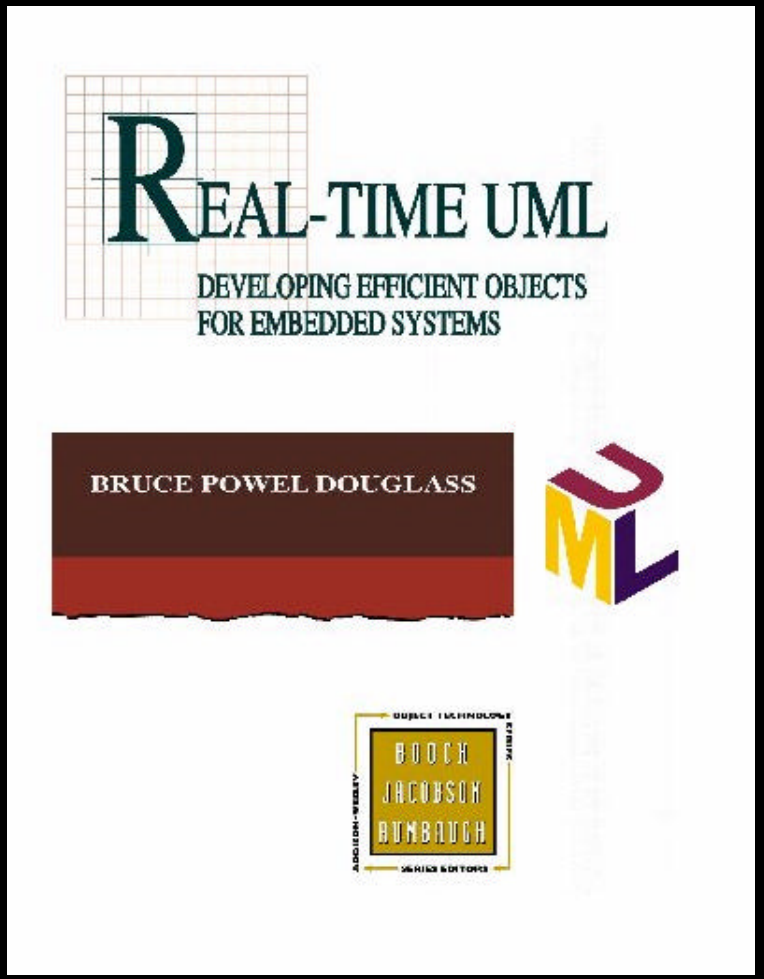
How to contact the author

Bruce Powel Douglass, Ph.D.
Chief Evangelist
i-Logix, Inc.
1309 Tompkins Drive Apt F
Madison, WI 53716
(608) 222-1056
bpd@ilogix.com

see our web site
www.ilogix.com

About the Author

- Almost 20 years in safety-critical hard-real time systems
- Mentor, trainer, consultant in real-time and object-oriented systems
- Author of
  - *Real-Time UML: Efficient Objects for Embedded Systems* (Addison-Wesley, Dec. 1997)
  - *Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time*
- Partner on the UML proposal
- Embedded Systems Conference Advisory Board



One man's "Of Course!" is another man's "Huh?"

Book of Douglass, Law 79

Feel free to ask questions!

Agenda

- Approach taken for this talk
- Types of behavior
- State Behavior
- Mealy-Moore State Models
- Harel Statecharts
- Integrating FSMs with your development process
- Producing code for state machines

Approach taken for this talk

- This is meant to be a gentle introduction to states and state machines
- This section will be mostly on state machines and a little on statecharts
- State models will first be introduced in simple forms
- Gradually concepts enhanced and elaborated
- Ask questions if you don’t think your neighbor is understanding

What kinds of things have Behavior?

- Objects!
  - Object have
    - Internal data
    - Operations on that data (behavior)
  - Objects can
    - react to environmental events and information
    - autonomously produce events and actions
  - Not all Objects have state behavior!

Types of Behavior

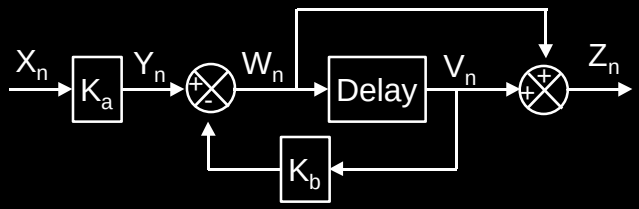
- Behavior can be *simple*
  Simple behavior does not depend on the object's history
- Behavior can be *continuous*
  Continuous behavior depends on the object's history but in a smooth, continuous fashion
- Behavior can be *state-driven*
  State-driven behavior means that the object's behavior can be divided into disjoint sets

Simple Behavior

- The behavior is not affected by the object's history
  - cos(x)
  - getTemperature( )
  - setVoltage(v)
  - max(a,b)
  - $\int_a^b e^{-x^2} \, dx$

Continuous Behavior

- Object's behavior depends on history in a continuous way
  - Control loops



  - digital filter

$$f_j = \frac{d_j + d_{j-1} + d_{j-2} + d_{j-3}}{4}$$

  - fuzzy logic

    Uses partial set membership to compute a smooth, continuous output

State Behavior

- Object exhibit discontinuous modes of behavior
  - in “therapy mode” delivery anesthetic agent based on knob position
  - in “service mode” select service function based on knob position
  - in “startup mode” ignore the knob turns

Why State Machines?

- Simplification
- Predictability
- Easy Development
- Easy Testing

Why: Simplification

- Introduce simplifying assumptions
  - Assumes system is only in a single state at a time
  - Assumes state transitions are instantaneous
- Limit interactions with other objects
  - A finite set of transitions are permitted while in any given state
  - Other events are
    - ignored
    - cause error recovery states to be entered
    - queued

Why: Simplification

- Overall behavior is decomposed into sets of non-overlapping behavior defined by
  - input events accepted while in a state
  - outputs initiated while in a state
  - error recovery mechanisms
- Easy Error Handling
  - easy to specify valid and non-valid inputs by in states

Why: Predictability

- FSMs divide their complexity into chunks called *states*
- Each state is simpler than the overall object
- Because each state is simpler, it is more understandable and predictable

Why: Easy Development

- FSMs simplify the system into smaller pieces
- Smaller pieces are easier to code
  - They are easy to write the code for
  - They are easier to “get right”
  - They are easy to explain in peer reviews

Easy Testing

- Unit testing is fundamentally at the level of the object (or function)
- FSMs can be subdivided into states for testing
  - Each state has a smaller set of input and output conditions
  - The overall testing is decreased because of low coupling among states

Finite State Machines

- A *finite state model* is the description from which any number of instances can be made
- A *finite state machine* (FSM) is an object which has state behavior defined by
  - A finite set of states
  - A finite set of transitions
- So,
  - What's a state?
  - What's a transition?

What is a *state*?

A state is a *distinguishable, disjoint, orthogonal ontological condition that persists for a significant period of time*

What is a *state*?

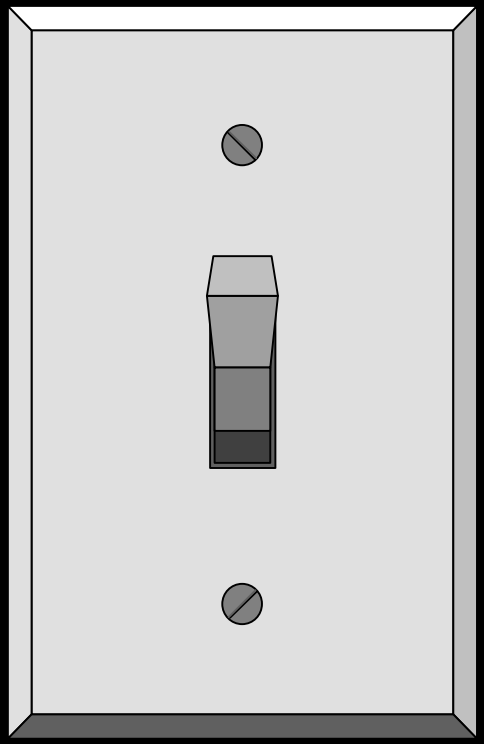
- Distinguishable
  It can be clearly distinguished from other states
  - Inputs accepted
  - Actions performed
- Disjoint
  An object can only be in one state in a time and must be in exactly one state at all times

What is a *state*?

- Orthogonal
  States do not overlap other states
- Ontological
  "fundamental condition of existence"
- Persists for a significant period of time
  - *Objects spend all their time in states*
    ("*Significant*" is problem-domain specific)

Example States: Switch

- State: Off
- State: On

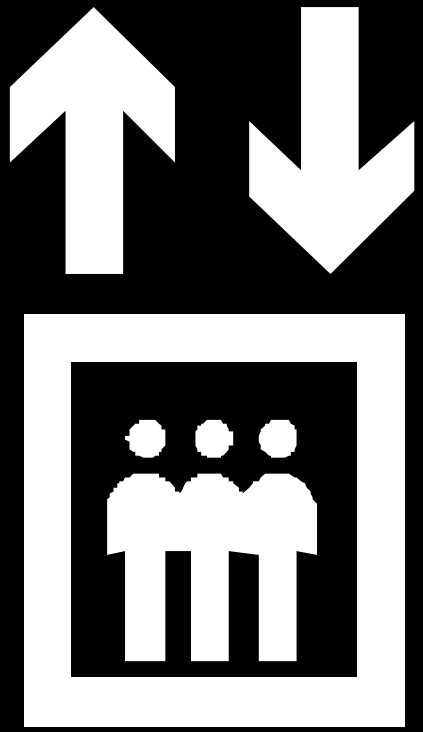


Example States: Oregon Weather

- State: Raining
- State: Going to Rain

Example States: Elevator

- State: Stopped
- State: Going Up
- State: Going Down



What constitutes a state?

- View 1: The value of all attributes of the object uniquely define the state
- View 2: The value of some specific attributes (state variables) uniquely defines the state
- View 3: A unique set of inputs accepted and actions performed defines the state

View 1: Values of All Attributes Defines the State

Sensor
float Value; enum tState s;

```
enum tState {Off,
    Calibrating,
    Measuring,
    NoValidMeasurement,
    ValidMeasurement }
```

- Are Value = 0.0 and Value = 0.000001 different states? YES!
- How many states are there? Infinite.
- Are the behaviors or events accepted in different states actually different? No.

View 2: Values of Some Set of Attributes Defines the State

Sensor
float Value; enum tState s;

```
enum tState {Off,
    Calibrating,
    Measuring,
    NoValidMeasurement,
    ValidMeasurement }
```

- Are Value = 0.0 and Value = 0.000001 different states? No.
- How many states are there? 5.
- Are the behaviors or events accepted in different states actually different? Yes.

View 3: Unique Set of Behaviors Defines the State

Sensor
float Value; enum tState s;

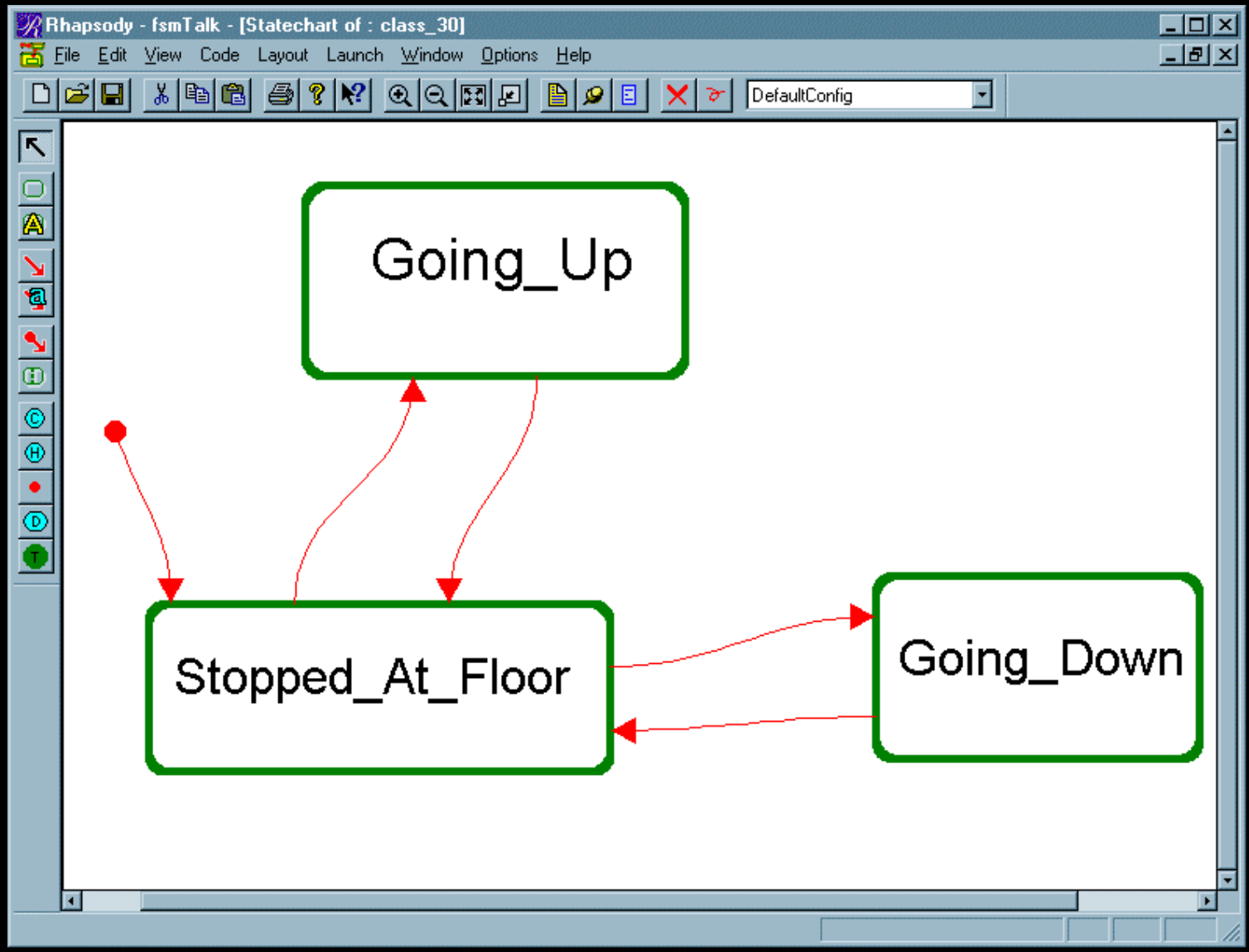
```
enum tState {Off,
    Calibrating,
    Measuring,
    NoValidMeasurement,
    ValidMeasurement }
```

- Are Value = 0.0 and Value = 0.000001 different states? No.
- How many states are there? 5.
- Are the behaviors or events accepted in different states actually different? Yes.

Transitions: Getting there is half the fun

- A *transition is the changing from one state of an object to another*
- Transitions are the FSM representation of *responses to events*
- Events may be from internal or external sources
- Transitions may have associated actions

Sample Transition: Elevator



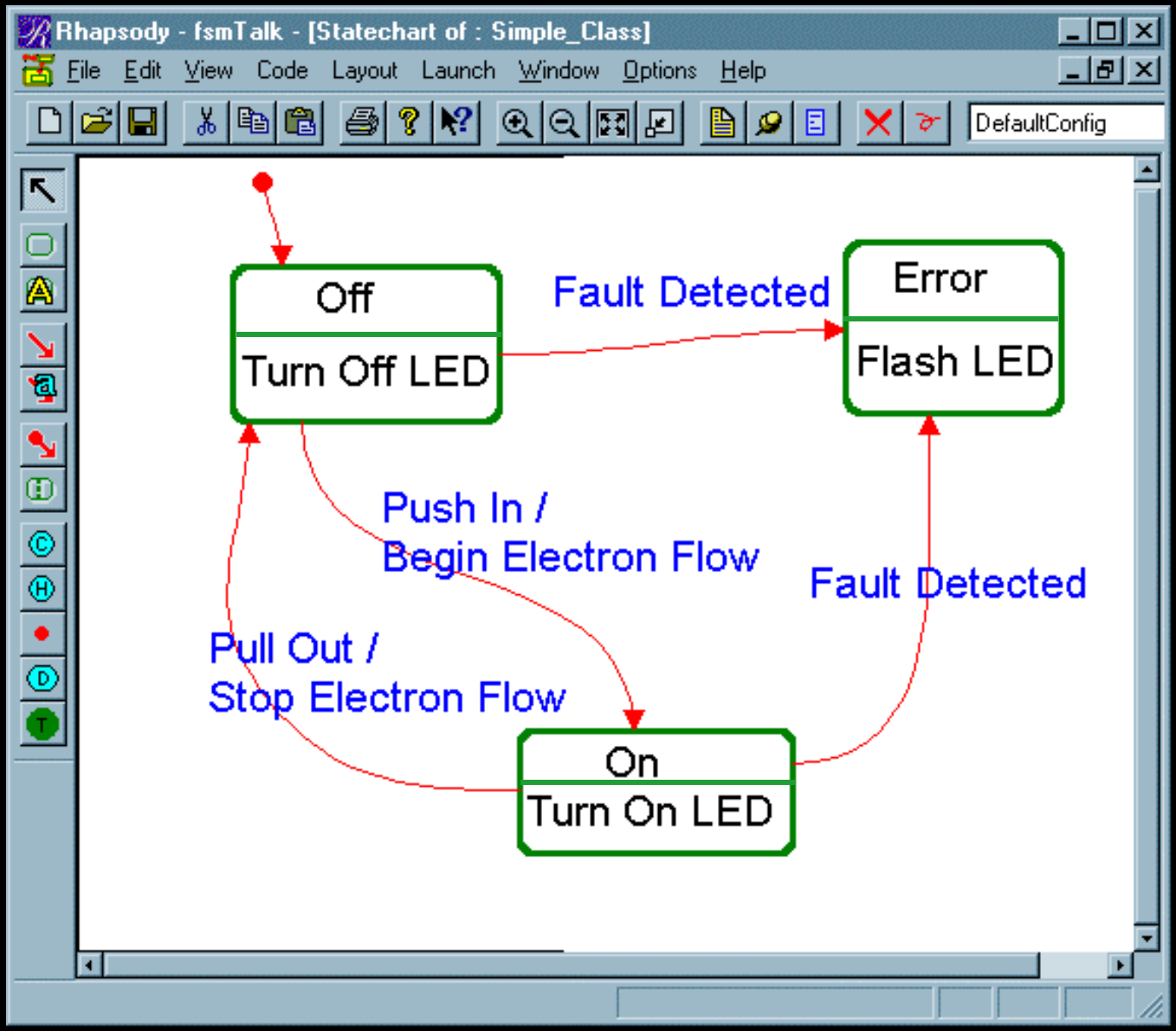
Actions

- Actions *are functions that take an insignificant amount of time to perform*
- Actions are implemented via
  - an object's operations
  - externally available functions
- They may occur when
  - A transition is taken
  - A state is entered
  - A state is exited

Actions

- Assign to a state when they are *always* executed on state entry or exit
- Assign to transition when they they are not always executed on state entry or exit

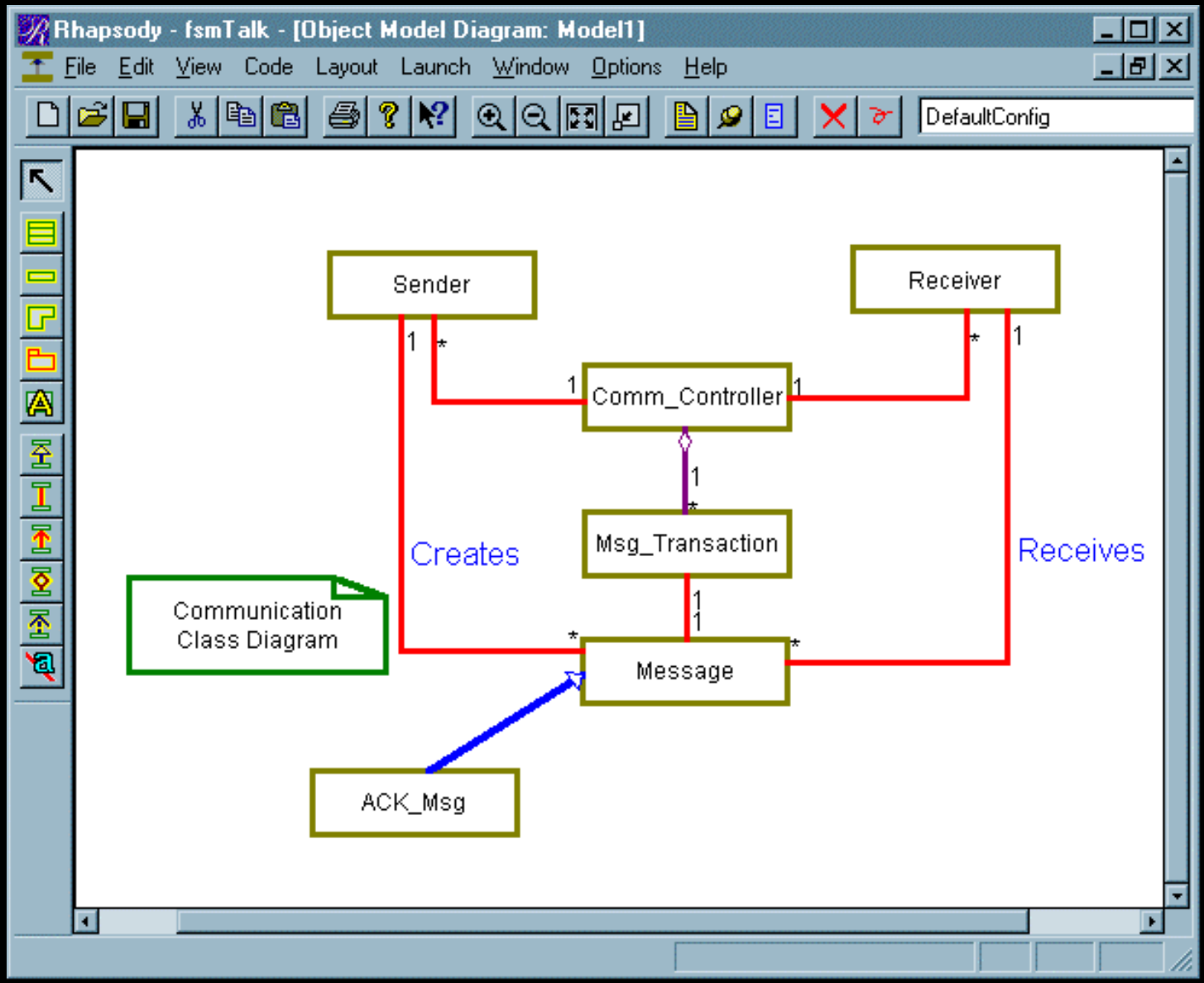
Simple FSM


Rhapsody - fsmTalk - [Statechart of : Simple_Class]
File Edit View Code Layout Launch Window Options Help
DefaultConfig
Off
Turn Off LED
Fault Detected
Error
Flash LED
Push In /
Begin Electron Flow
Fault Detected
Pull Out /
Stop Electron Flow
On
Turn On LED

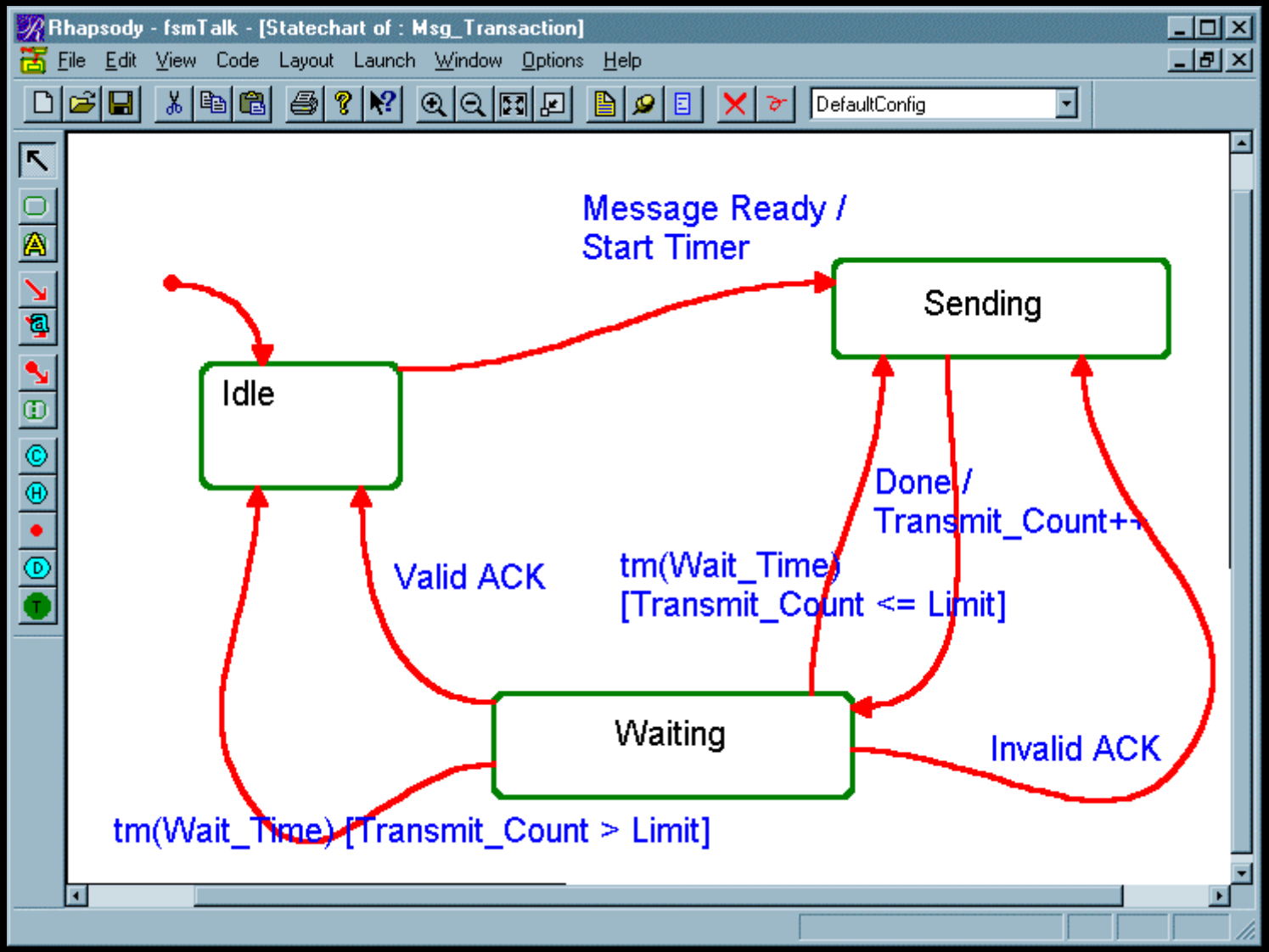
A Slightly More Complex FSM

- You are implementing a reliable transmission service for an OSI-compliant protocol stack.
- A message is sent that requires the receiver to return an ACK.
- If an ACK does not occur, retransmit the message
- If the message is transmitted 5 times without an ACK, then inform the sender.

What's the Object?



Message Transaction FSM



Mealy-Moore State Models

- The set of states defines the *state space*
- State spaces are flat
  - All states are at the same level of abstraction
  - All state names are unique
- State models are single-threaded
  - Only a single state can be valid at any time

Mealy-Moore State Models

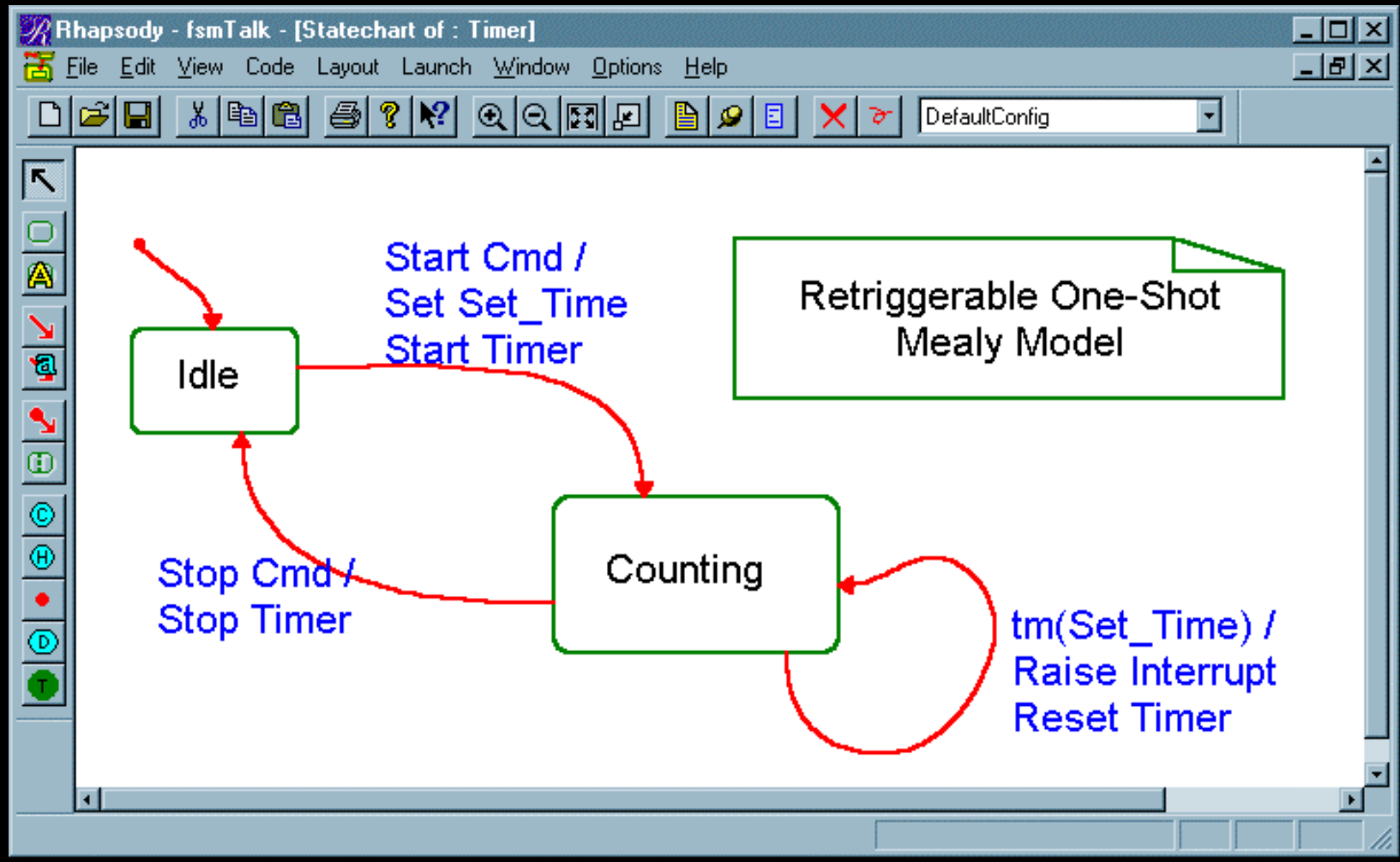
- Mealy State Models
  - All actions are in transitions
- Moore State Models
  - All actions are upon state entry

Retriggerable One-shot Timer FSM

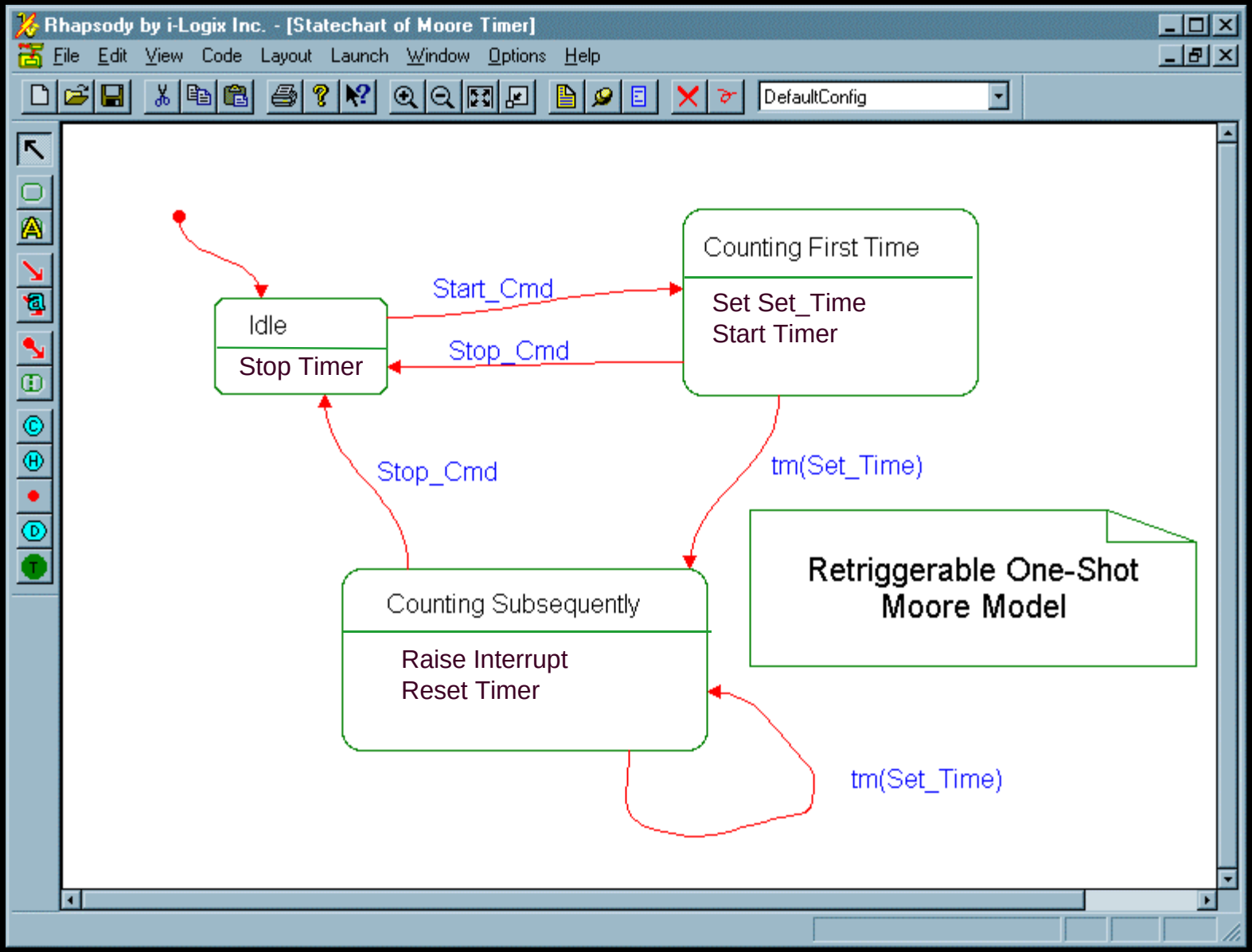
- How many states?
- Model 1
  - Idle
  - Count = 65,535
  - Count = 65,534
  - ...
  - Count = 0
- Model 2
  - Idle
  - Counting down

ANS: Model 2
(Drawing model 1 is a homework exercise)

Retriggerable One-shot Timer



Retriggerable One-Shot Timer

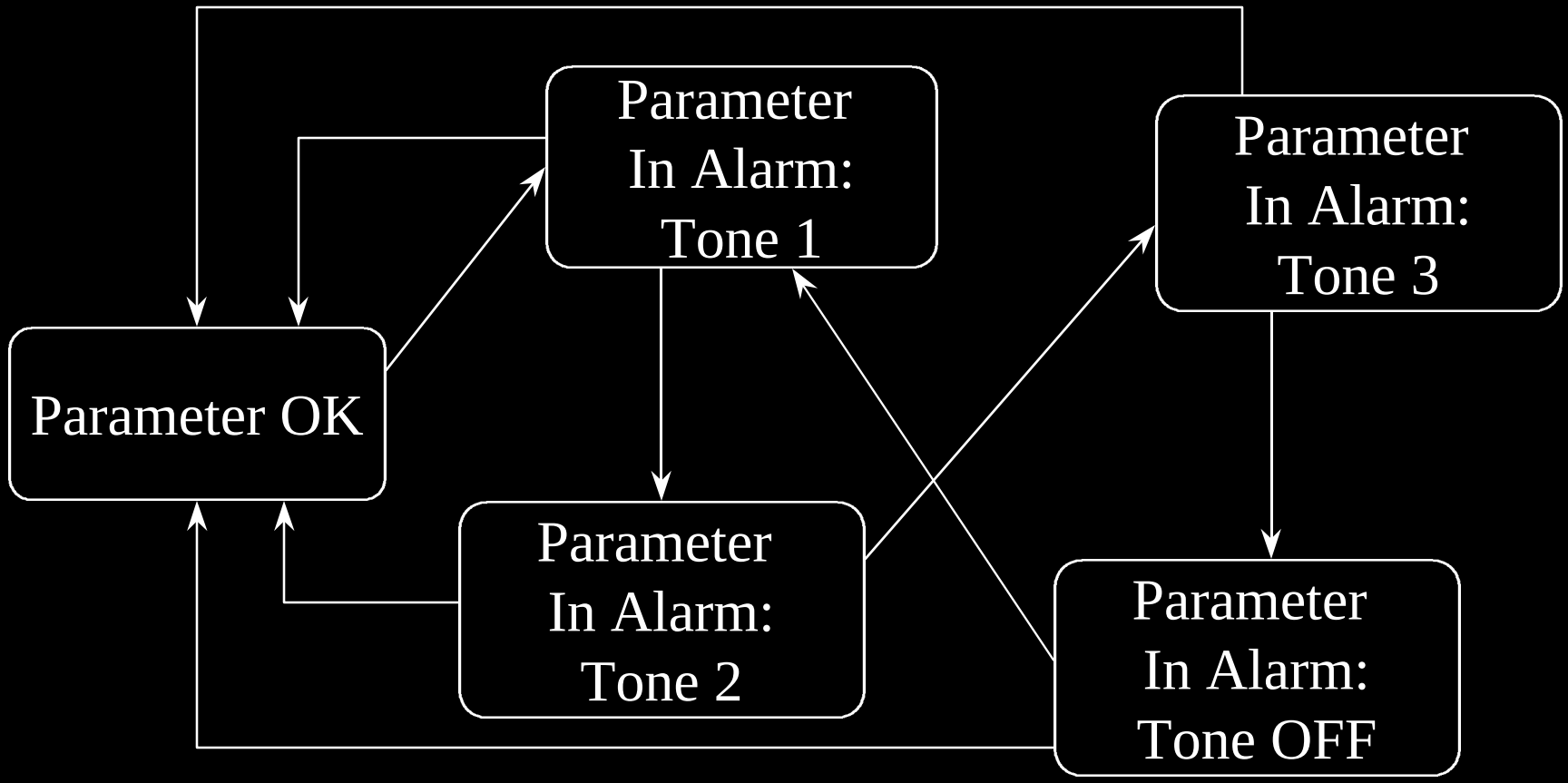


Problems with M&M State Models

- Scalability due to lack of metaphor for decomposition
- No Concurrency support
- No support for orthogonal components

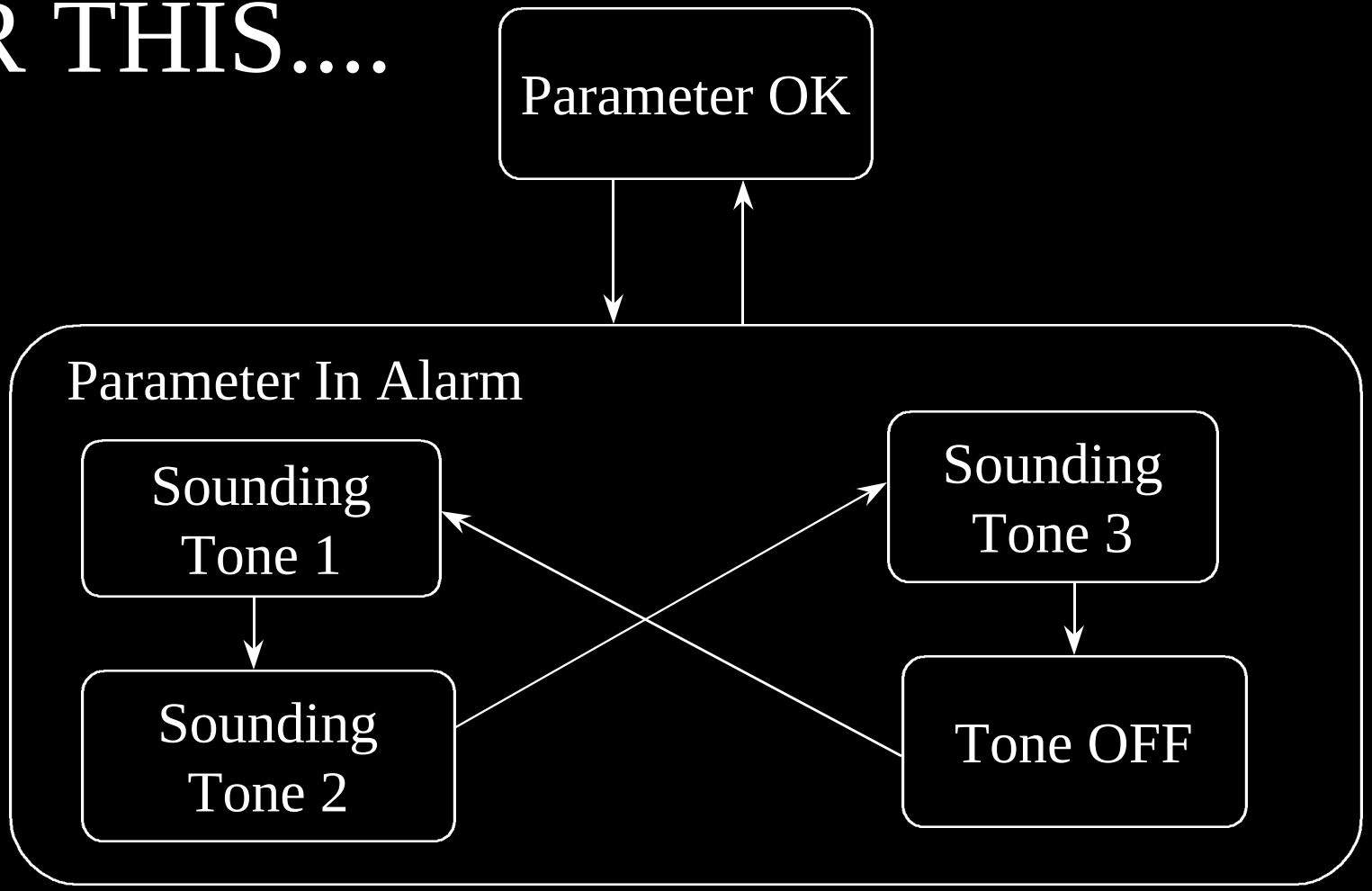
Scalability

THIS....



Scalability

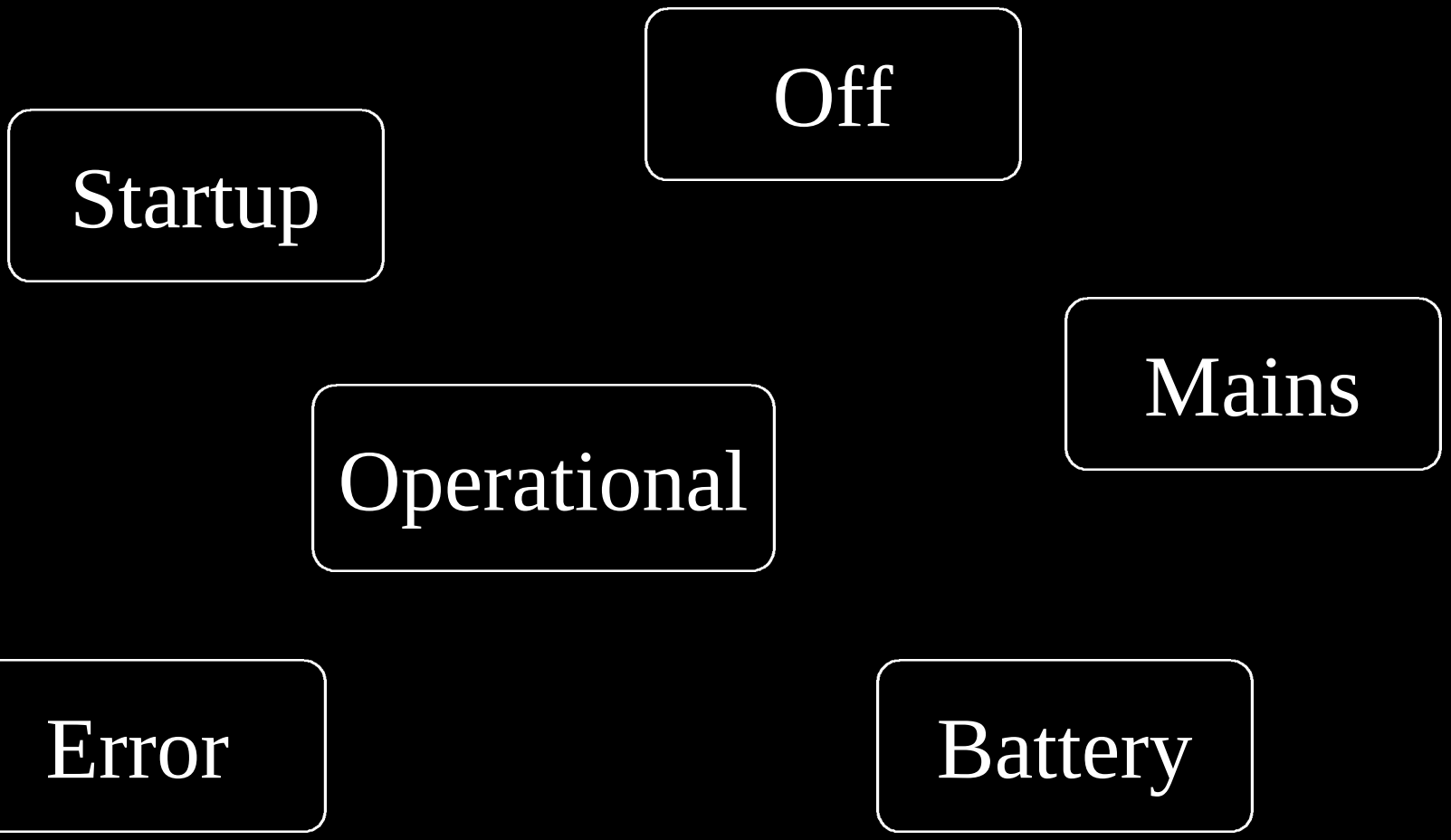
OR THIS....



Concurrency

- Problem: A device can be in states
  - Off
  - Starting Up
  - Operational
  - Error
- And it can be running from
  - mains
  - battery

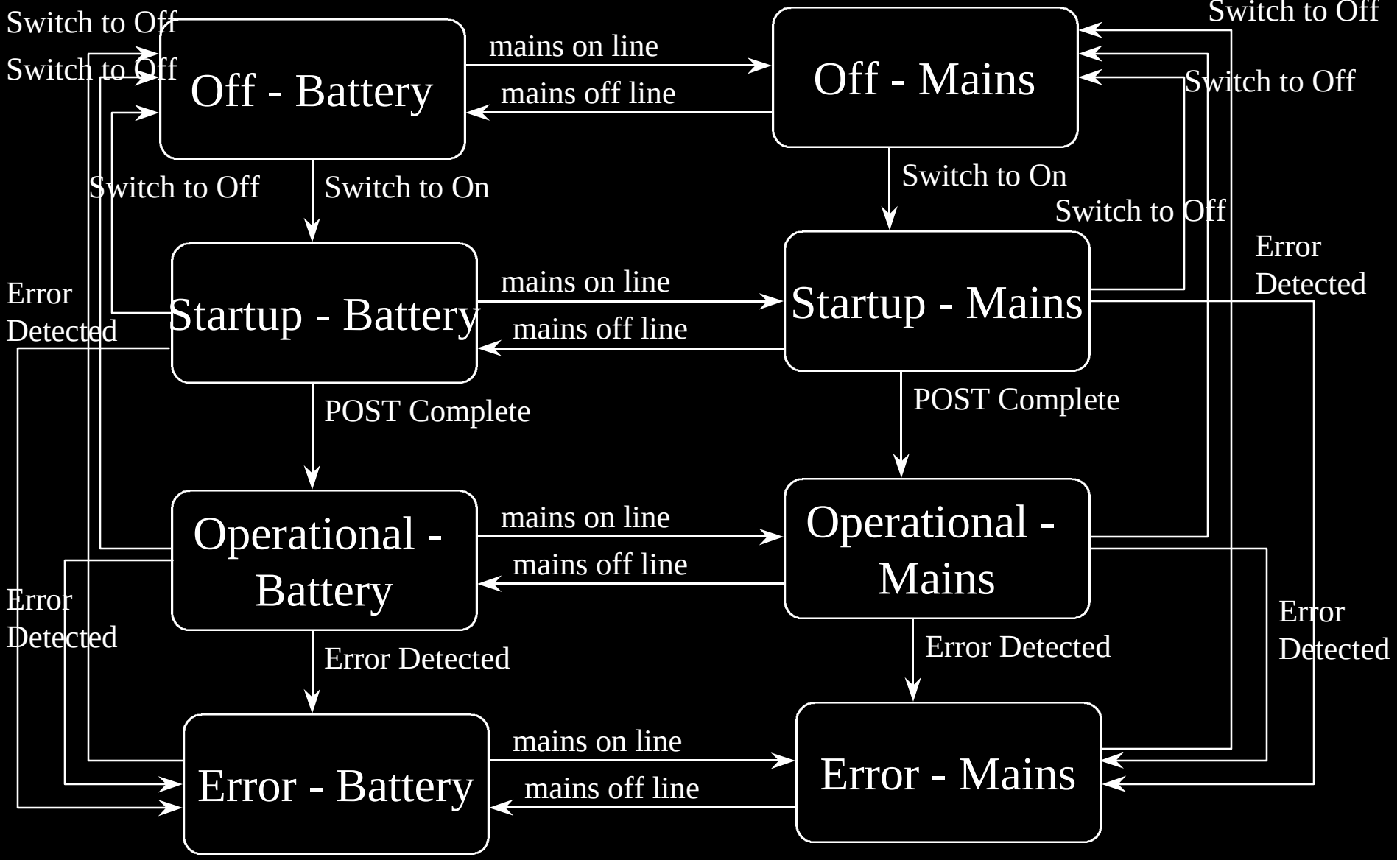
How to arrange these states?



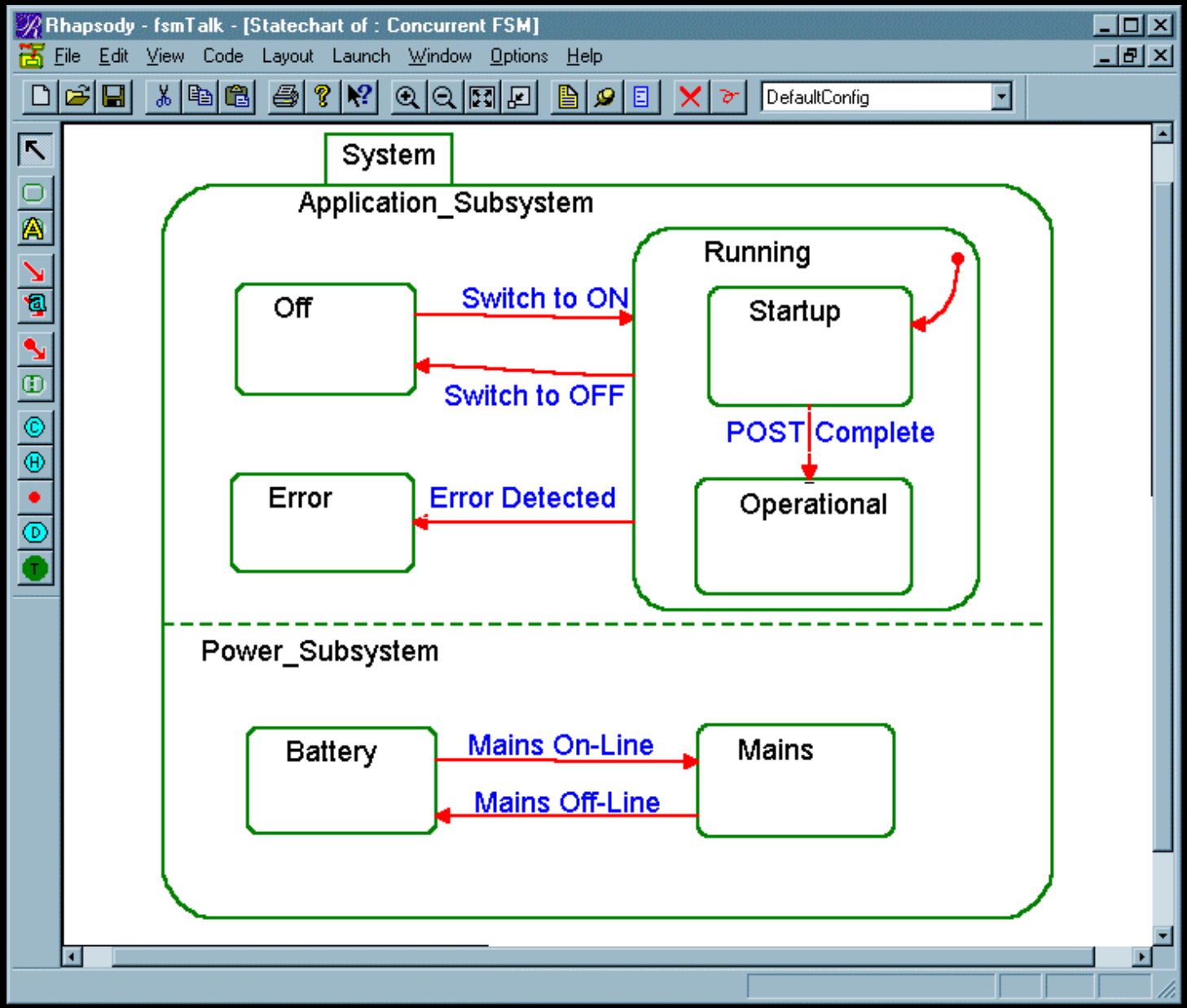
Concurrency

- In M&M View, the following are different states
  - Operational with Battery
  - Operation with Mains
- This is called *state explosion*
- Solution:
  - Allow states to operate concurrently

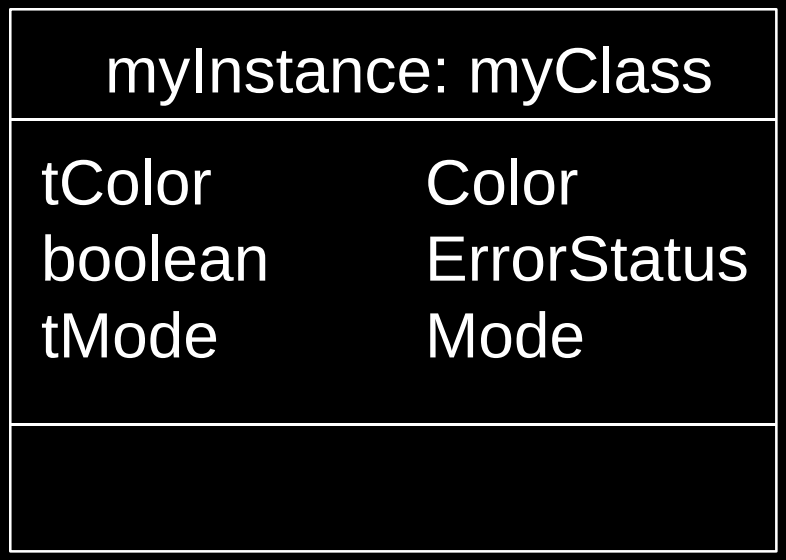
Mealy-Moore Solution



Concurrent State Model Solution



Orthogonal Components



enum tColor {eRed, eBlue, eGreen};

enum boolean {TRUE, FALSE}

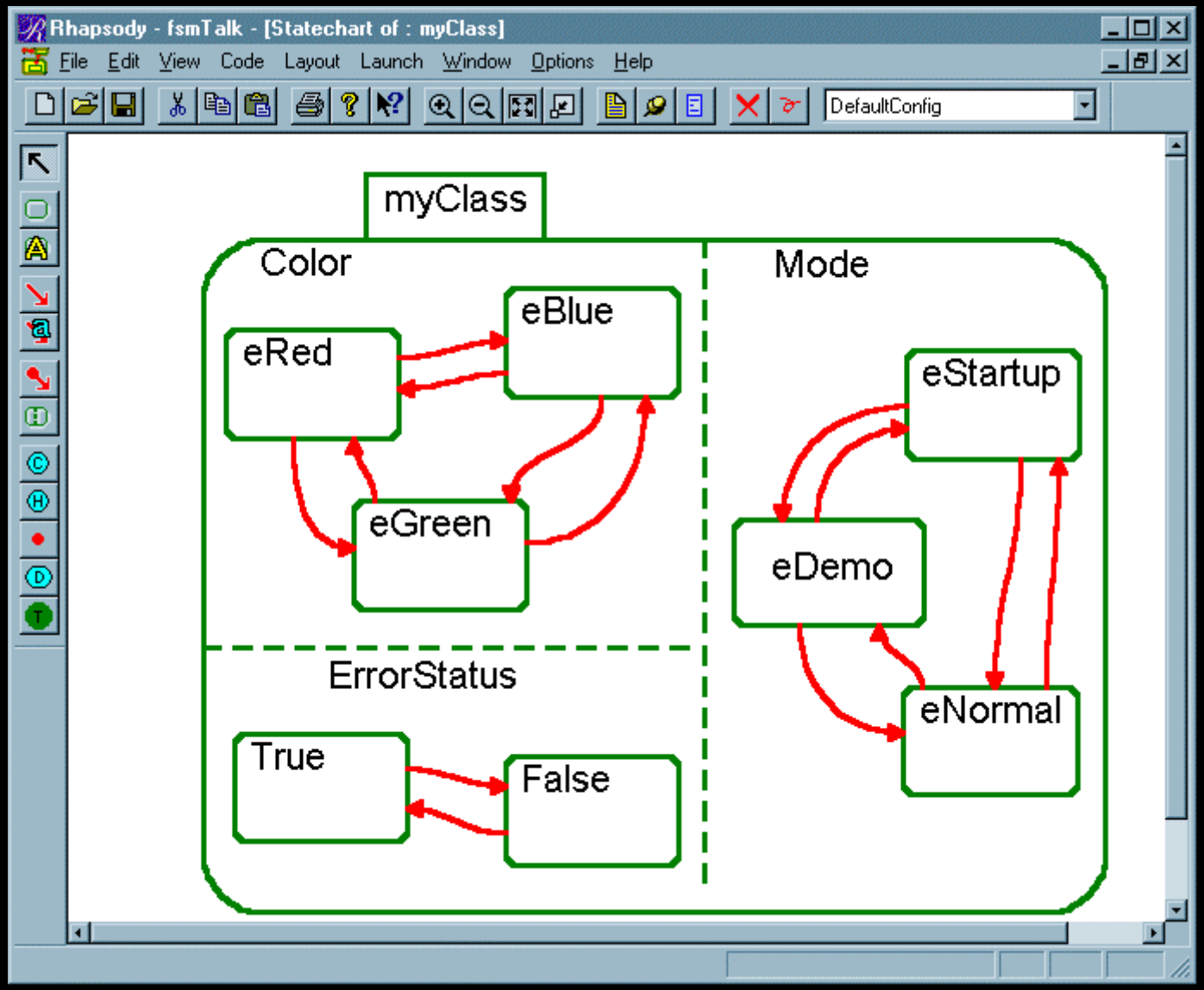
enum tMode {eNormal, eStartup, eDemo}

How do you draw the state of this object?

Approach 1: Enumerate all

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

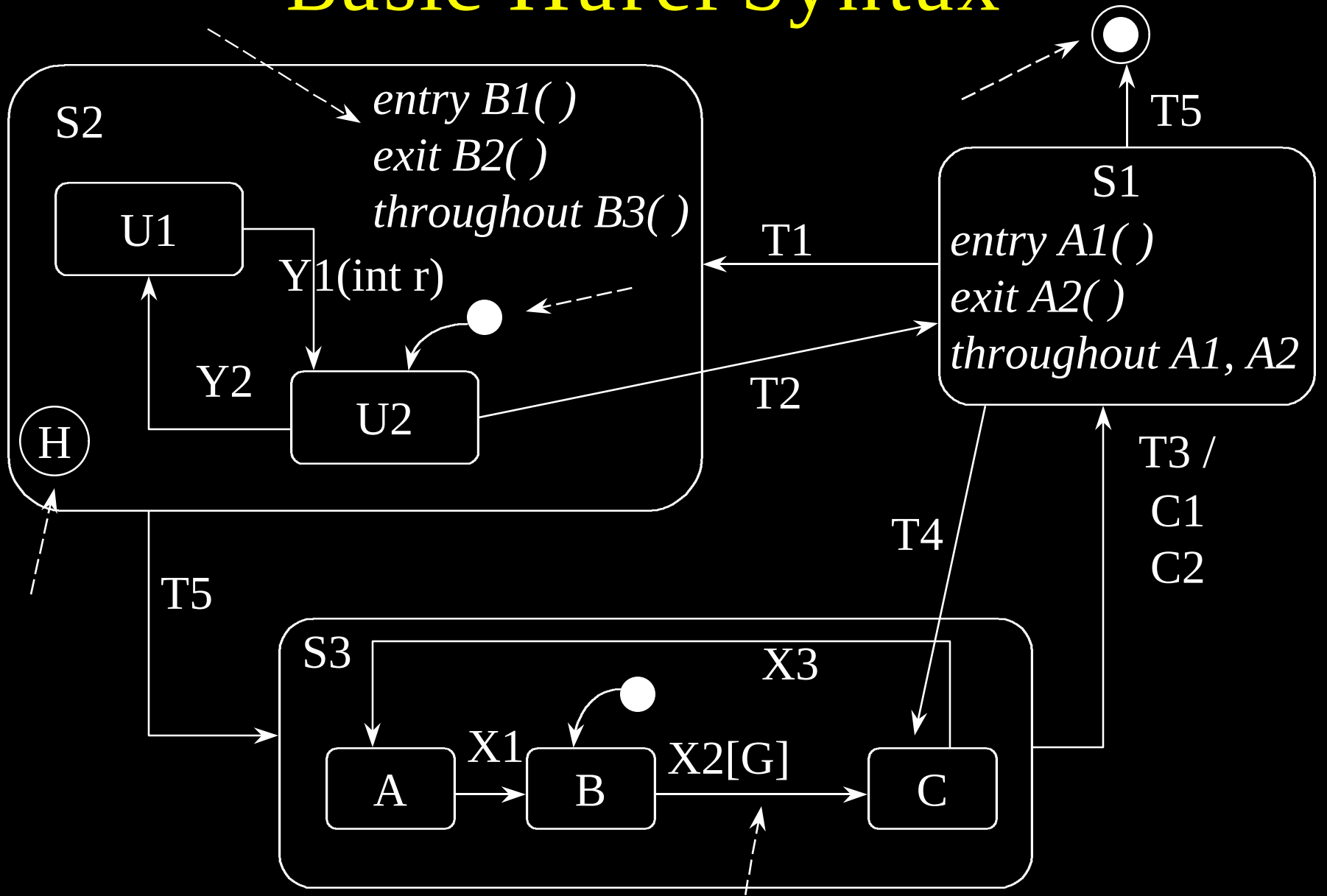
Approach 2


Rhapsody - fsmTalk - [Statechart of : myClass]
File Edit View Code Layout Launch Window Options Help
DefaultConfig
myClass
Color
eBlue
eRed
eGreen
Mode
eStartup
eDemo
eNormal
ErrorStatus
True
False

Harel State Charts

- Created by David Harel in late 1980s
- Supports
  - Nested states
  - Actions and Activities
  - Guards
  - History
- Advanced Features (Part II)
  - Concurrency
  - Broadcast Transitions
  - Orthogonal Components

Basic Harel Syntax



Nested States

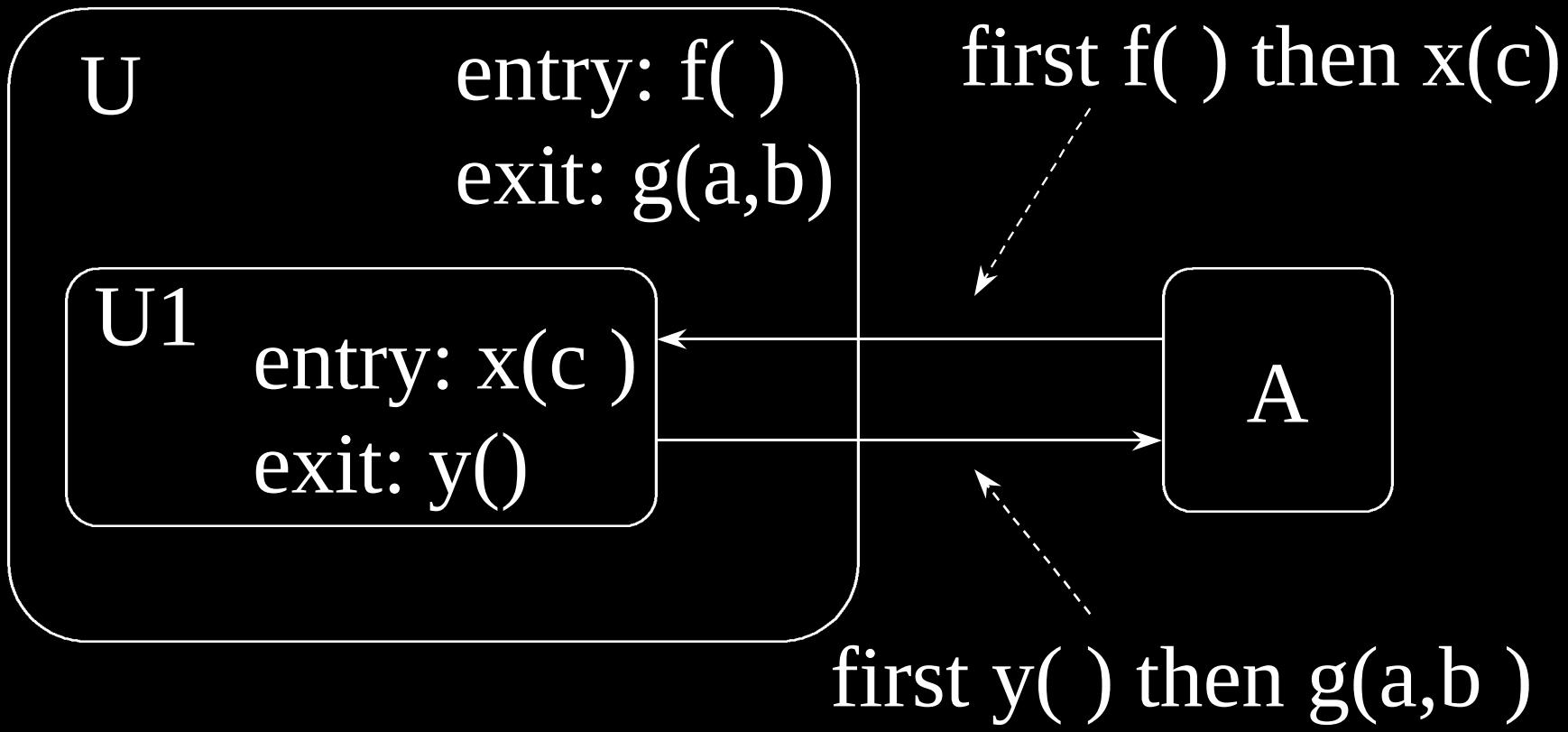
- Improves scalability
- Increases understandability
- Permits problem decomposition (*divide-and conquer*)
- Methods
  - Nested states on same diagram
  - Nested states on separate diagram

Actions and Activities

- Actions
  - are functions which take an insignificant amount of time to execute
  - they may have parameters
  - they may occur on
    - Transitions
    - State Entry
    - State Exit
- Activities
  - are functions executed as long as a state is active

Order of Nested Actions

- Execute from outermost - in on entry
- Execute from innermost - out on exit



Transitions

- Basic (UML) syntax:

  name(params)[guards]^events/actions

  - Name
  - Parameters
  - Guard
  - Event List
  - Action List

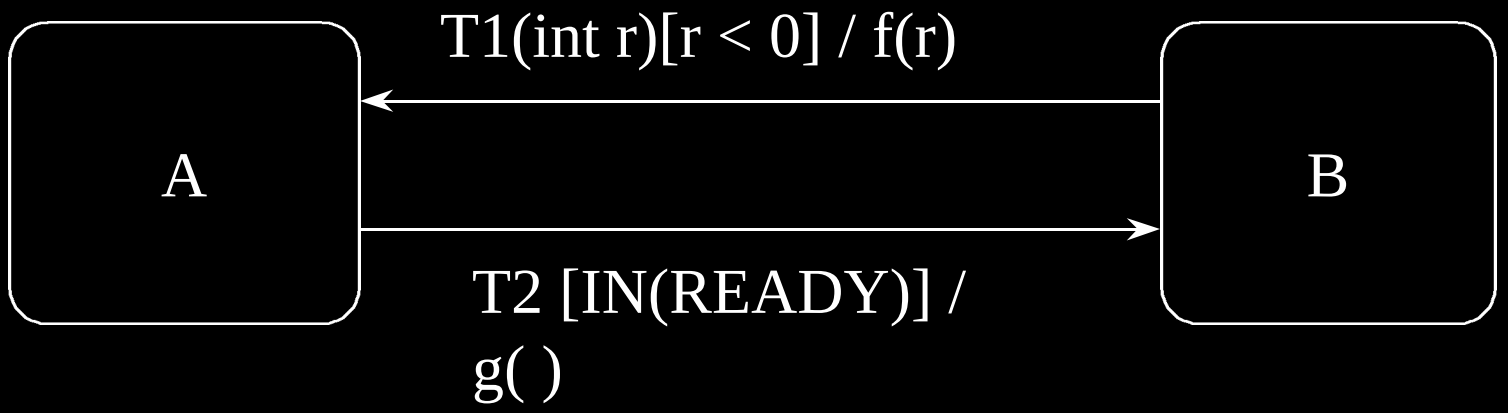
Transitions: Event List

- Comma separated list of transitions that occur in other concurrent state machines because of this transition
- A.k.a *propagated transitions*
- This will be discussed more in Part II

Transitions: Guards

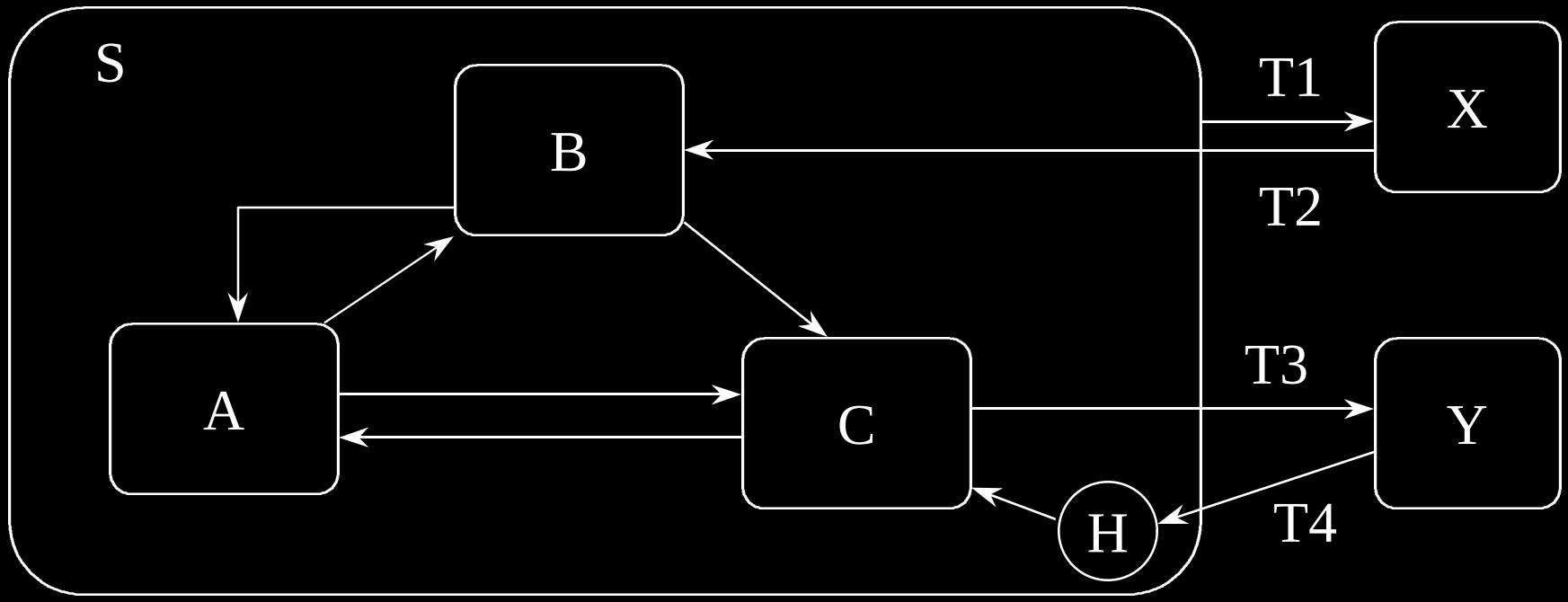
- A guard is some condition that must be met for the transition to be taken
- Guards can be
  - Variable range specification
  - Concurrent state machine is in some state [IN(X)]
  - Some other constraint (*preconditional invariant*) must be met

Transitions

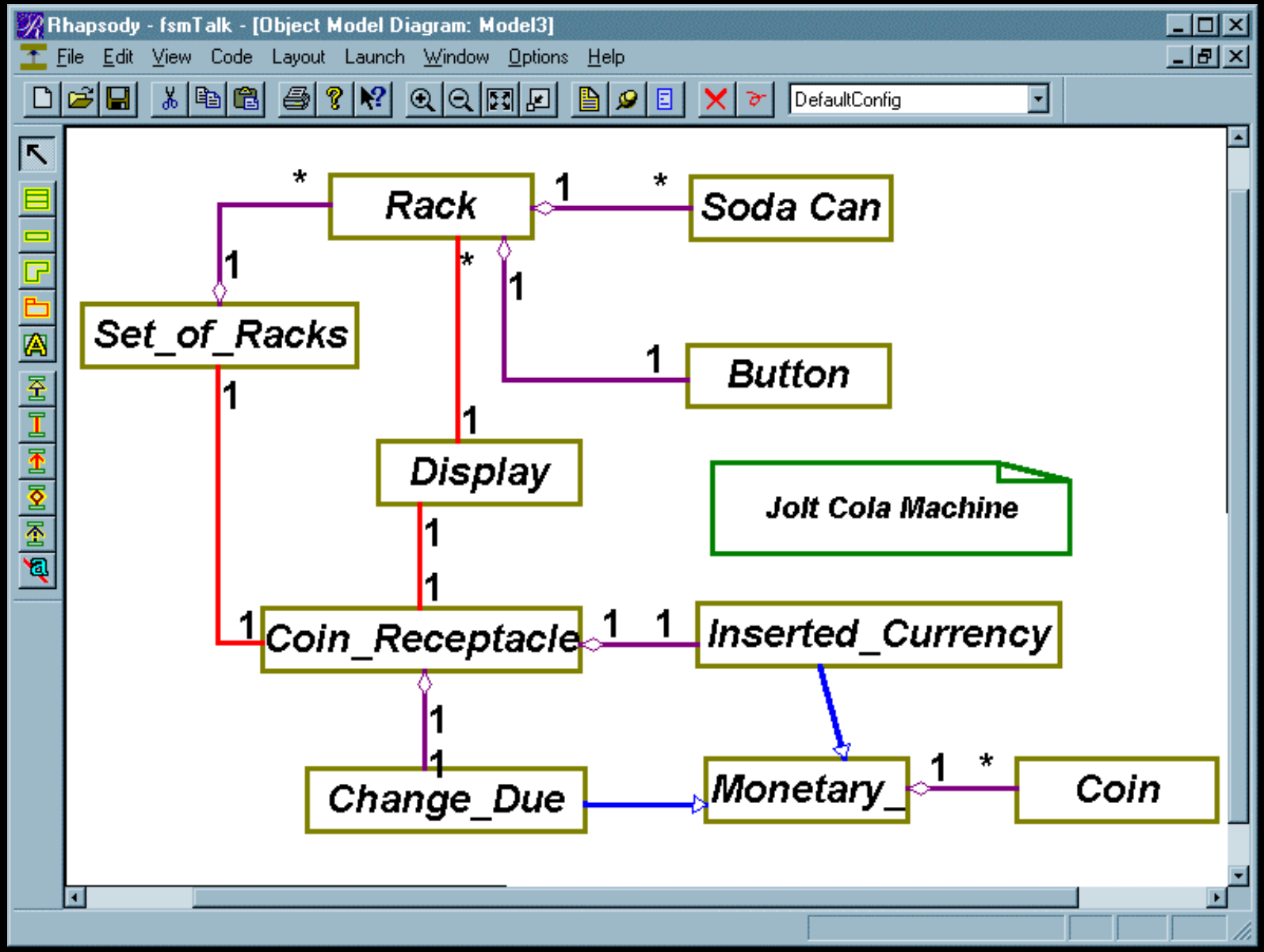


History

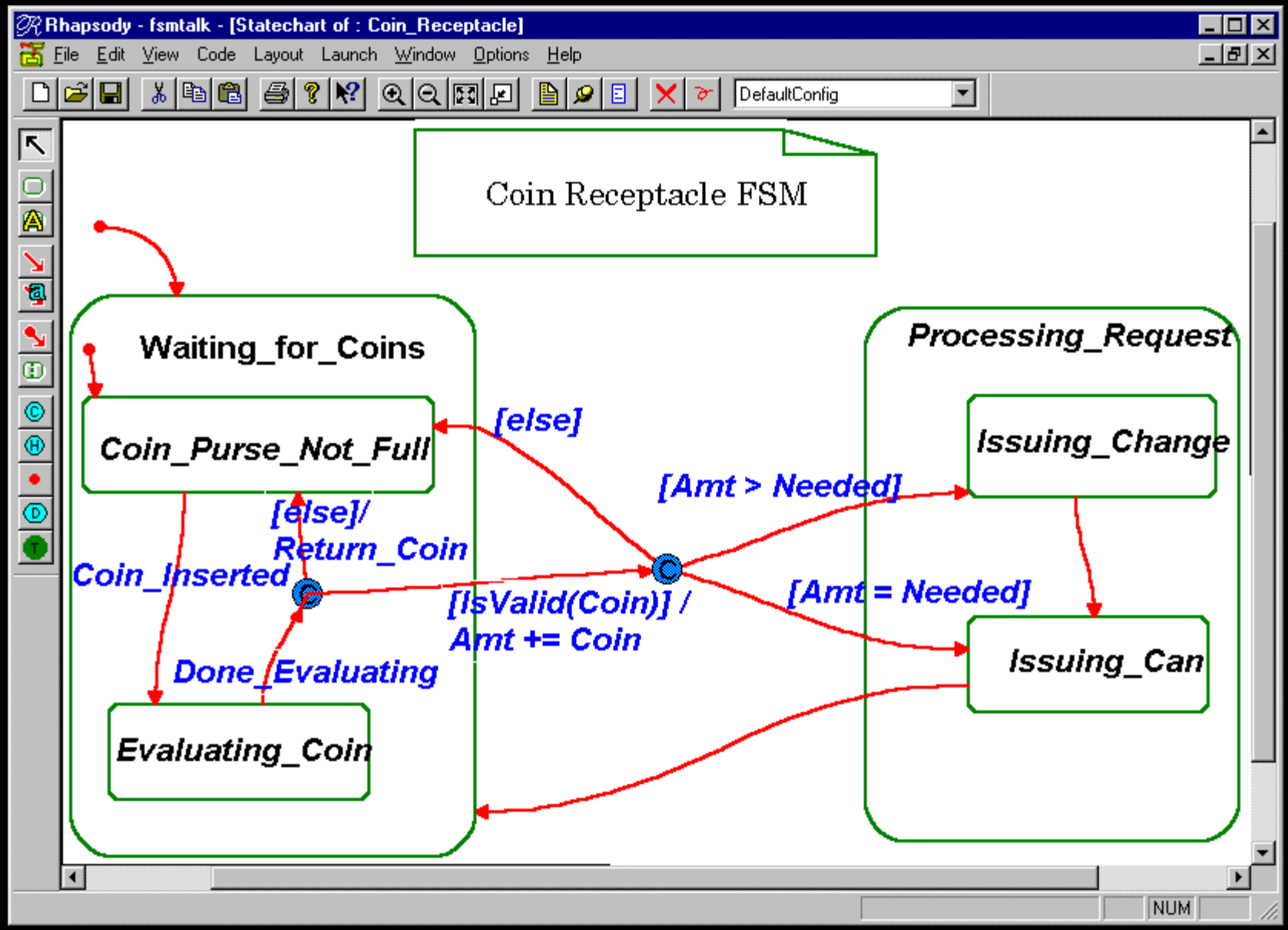
- The history annotation (H) means that the state “remembers” the substate and returns to it as the default
- Can also work with an initial state indicator



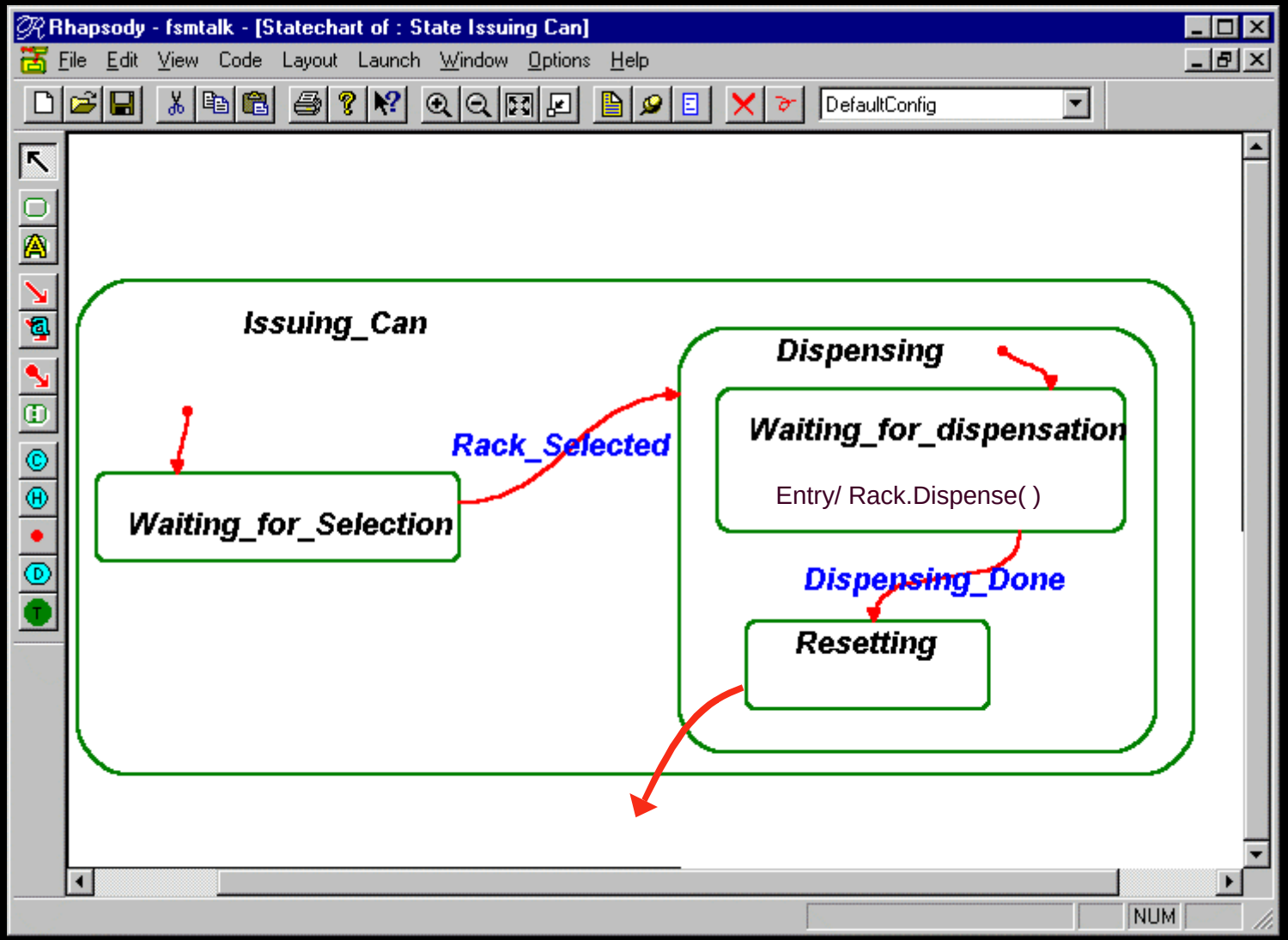
Example: Jolt Cola Machine



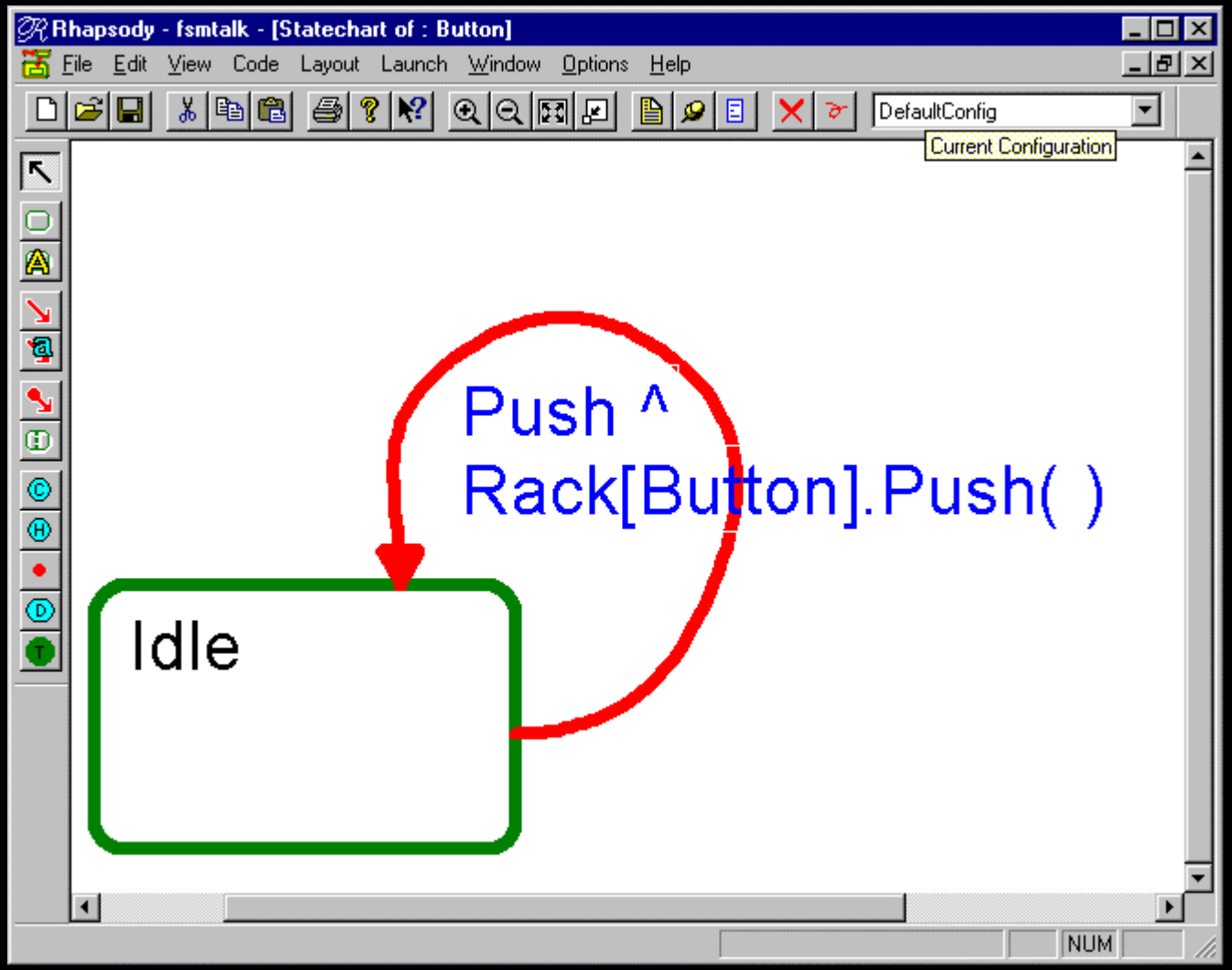
Example: Coin Receptacle FSM



Substate: Issuing Can



Class Button FSM



Summary

- Objects have behavior
  - Simple
  - Continuous
  - State-driven
- Modeling objects as Finite State Machines simplifies the behavior
- States apply to objects
- FSM Objects spend all their time in exactly 1 state (discounting orthogonal substates)

Summary

- States are *disjoint ontological conditions that persist for a significant period of time.*
- States are defined by one of the following:
  - The values of all attributes of the object
  - The values of specific attributes of the object
  - Disjoint behaviors
    - Events accepted
    - Actions performed

Summary

- Transitions are the representation of responses to events within FSMs
- Transitions take an insignificant amount of time
- Actions are functions which may be associated with
  - Transitions
  - State Entry
  - State Exit
- Activities are processing that continues as long as a state is active

Summary

- Harel statecharts expand standard FSMs
  - Nested states
  - Concurrency
  - Broadcast transitions
  - Orthogonal Components
  - Actions on states or transitions
  - History
  - Guards on transitions


i-Logix
Embedded Systems and Software Design Automation