State Machines and Statecharts

Part 2

State Machines

Bruce Powel Douglass, Ph.D.

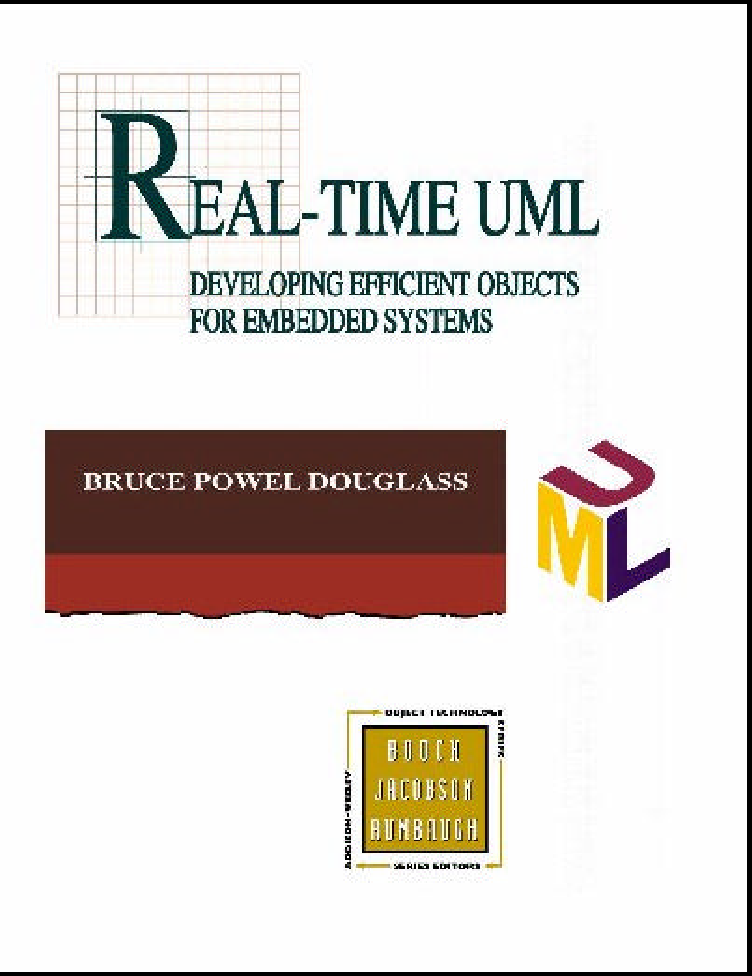
How to contact the author

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



All the best lies are actually true!

Agenda

- Approach taken for this talk
- Quick Overview of Finite State Machines
- Quick Overview of Harel Statecharts
- Advanced Statechart features
- Other State Notations

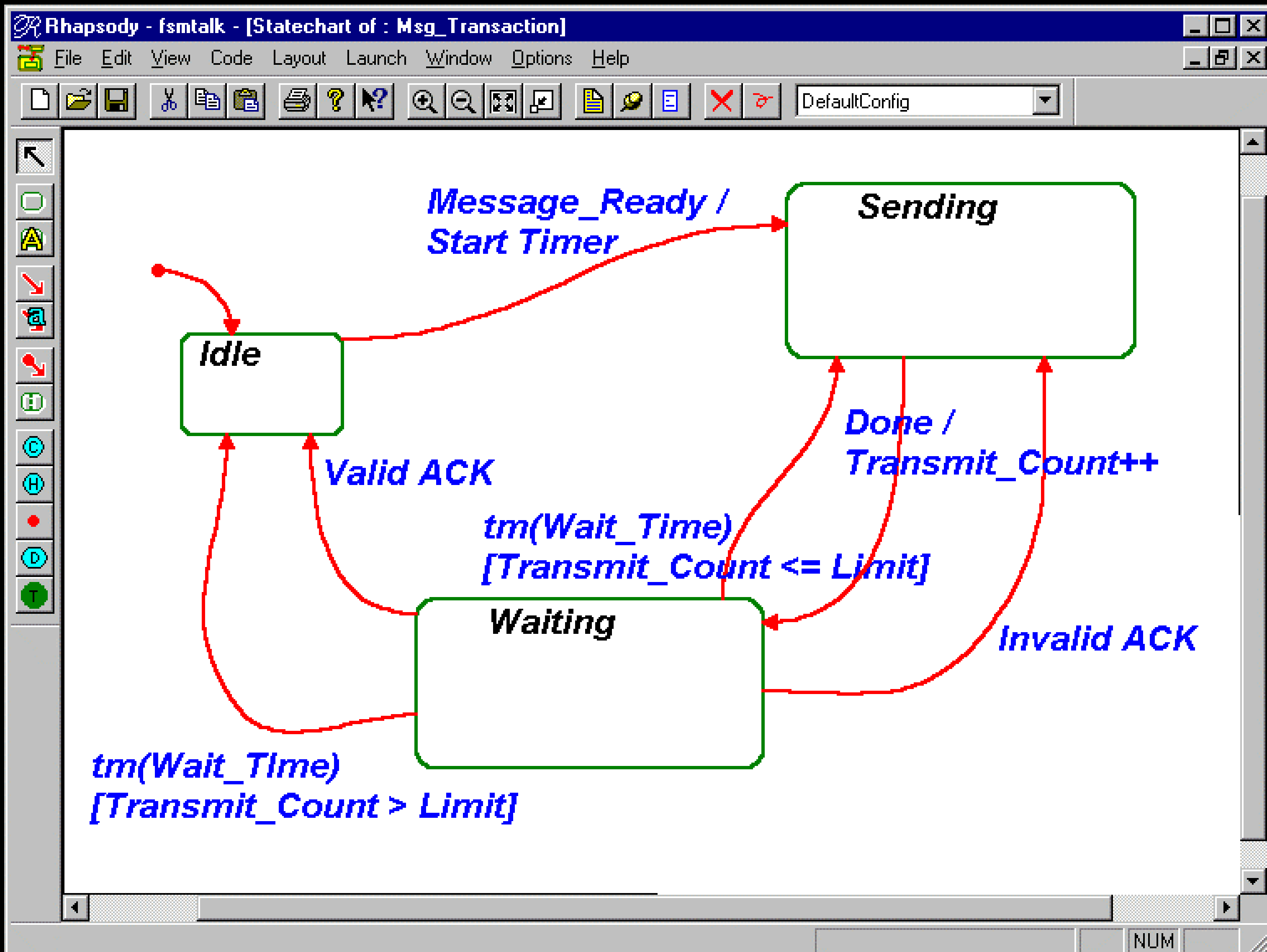
Approach taken for this talk

- This is meant to be a gentle introduction to states and state machines
- This section will be
  - mostly on advanced features of statecharts
  - other state representations
- Ask questions if you don't think your neighbor is understanding

Finite State Machine Review

- What's a STATE?
- What's a TRANSITION?
- What are the three classes of behavior?
- What kinds of things have state?
- Why model states?

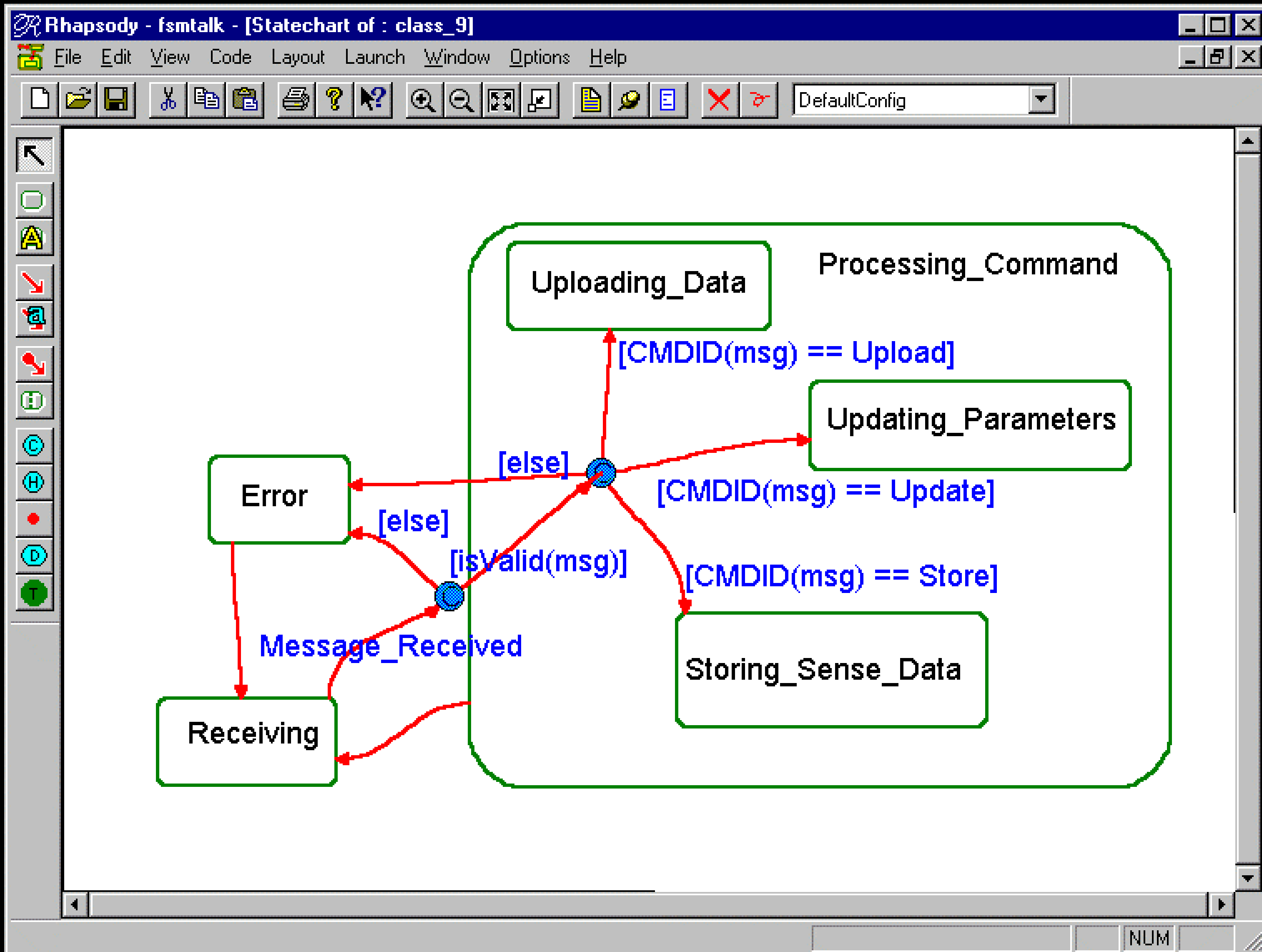
Simple Example



Advanced Statechart Features

- Conditional Transitions
- Orthogonal Components
- Concurrency
- Broadcast transitions
- Inherited state behavior

Conditional Transitions



Orthogonal Components

- Model state behavior for independent aspects of objects
- Can be used to model
  - concurrency
  - independent attributes
- Simplify state diagrams by reducing "state explosion"

Orthogonal Components

myInstance: myClass	
tColor	Color
boolean	ErrorStatus
tMode	Mode

enum tColor {eRed, eBlue, eGreen};

enum boolean {TRUE, FALSE}

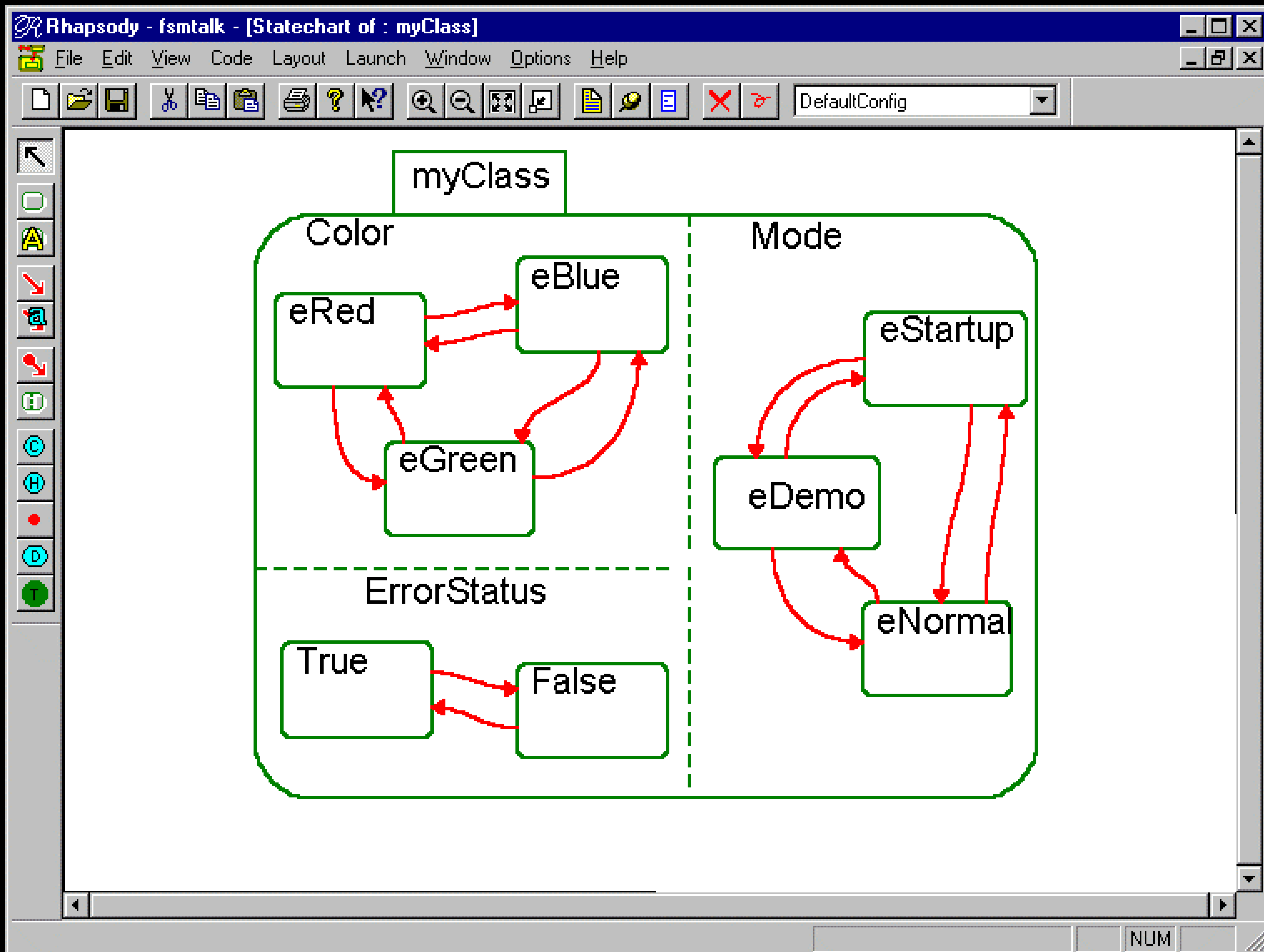
enum tMode {eNormal, eStartup, eDemo}

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
eRed
eBlue
eGreen
ErrorStatus
True
False
Mode
eStartup
eDemo
eNormal
NUM

What *is* Concurrency?

- Concurrency is *the simultaneous execution of program statements within a system*
- Types:
  - Pseudo-concurrency (Single CPU)
  - True Concurrency (Multiple CPUs)

Pseudo-Concurrency

- Heavy-weight (process)
  Each process has its own data and code space

- Light-weight (thread)
  Each thread shares a common data and code space

Synchronization Models

- Sharing data
  - Shared variables
  - Message passing
- Types of synchronization
  - Synchronous
  - Asynchronous
  - Balking
  - Timeout

Synchronization Models

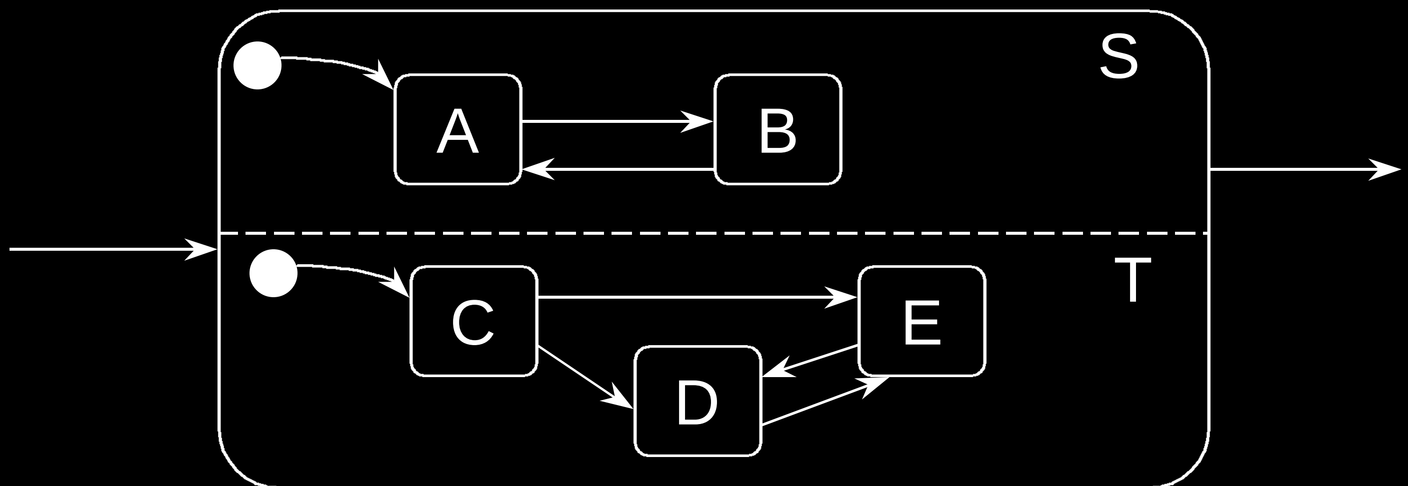
- Operations may be
  - Guarded
  - Synchronous
  - Simple (i.e. function calls)
- Events imply
  - Asychronicity
  - Event queues

UML Concurrency

- Each thread is based from a single “active” object
- All components of the active object inherit the composite’s thread
- Each thread must have its own event queue

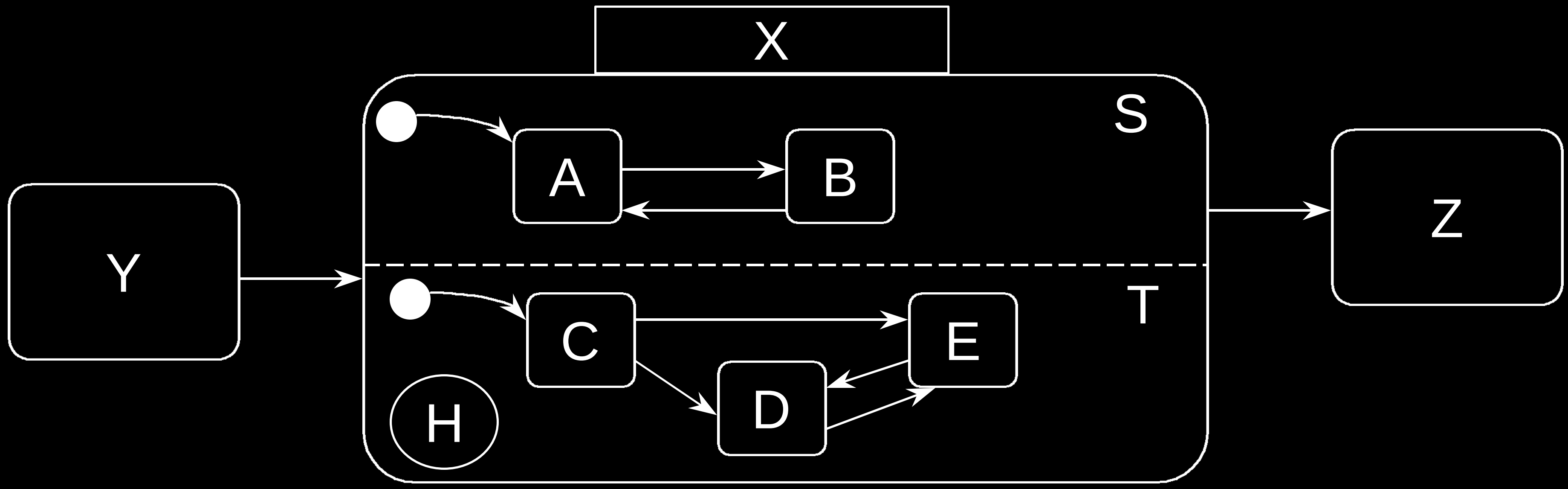
Concurrent Statecharts

- Many embedded systems consist of multiple threads, each running an FSM
- State charts allow the modeling of these parallel threads



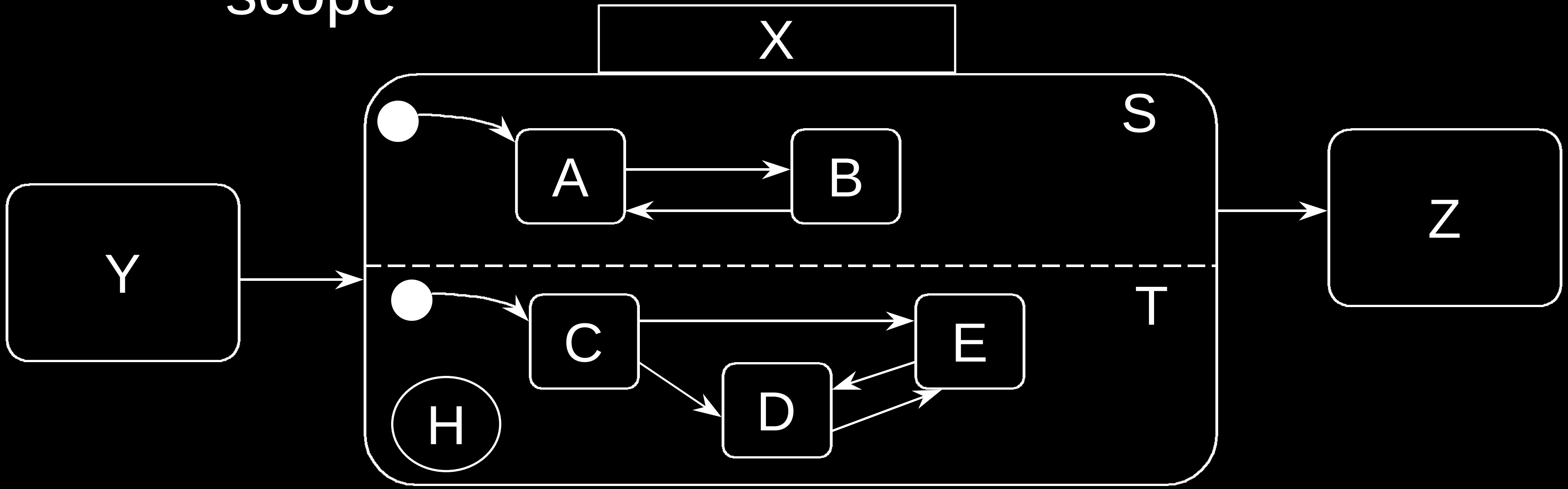
Concurrent State Charts

- States S and T are active at the same time as long as X is active.
  – Either S.A or S.B must be active when S is active
  – Either T.C, T.D, or T.E must be active when T is active

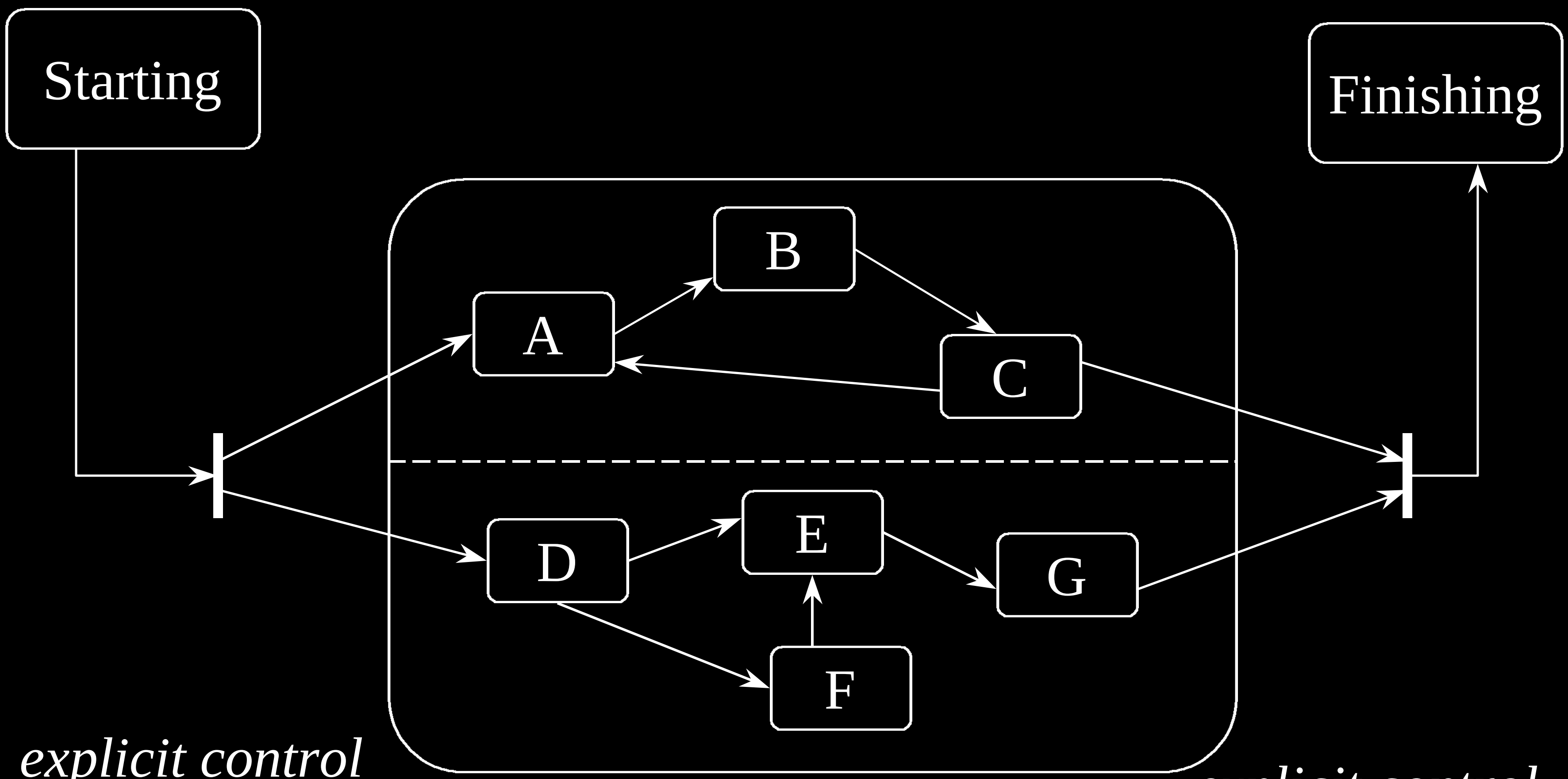


Concurrent State Charts

- When X exits, both S and T exit
  - If S exits first, the FSM containing X must wait until T exits
  - If the two FSMs are always independent, then they must be enclosed at the highest scope



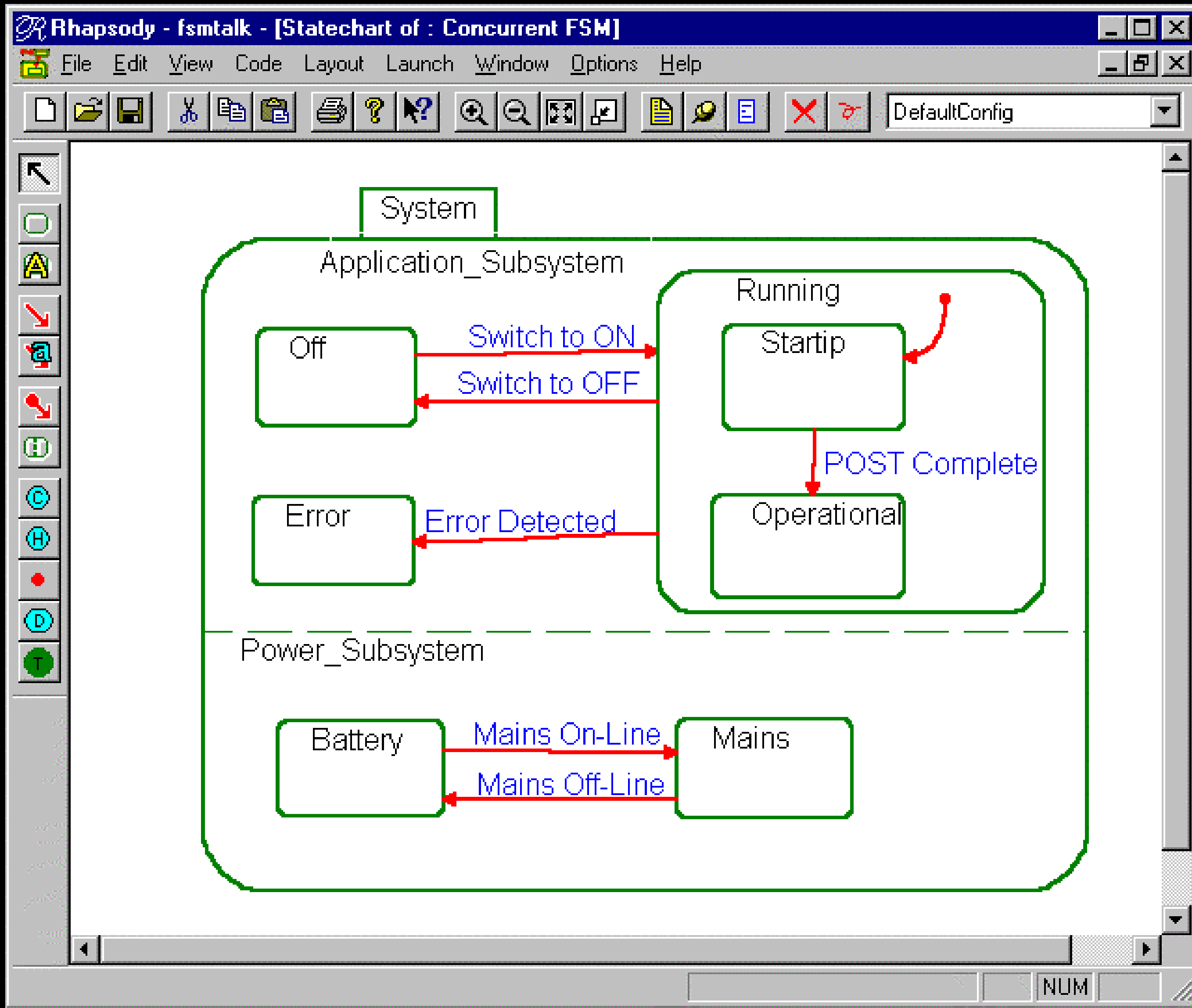
Explicit Synchronization



explicit control branching

explicit control resynchronization

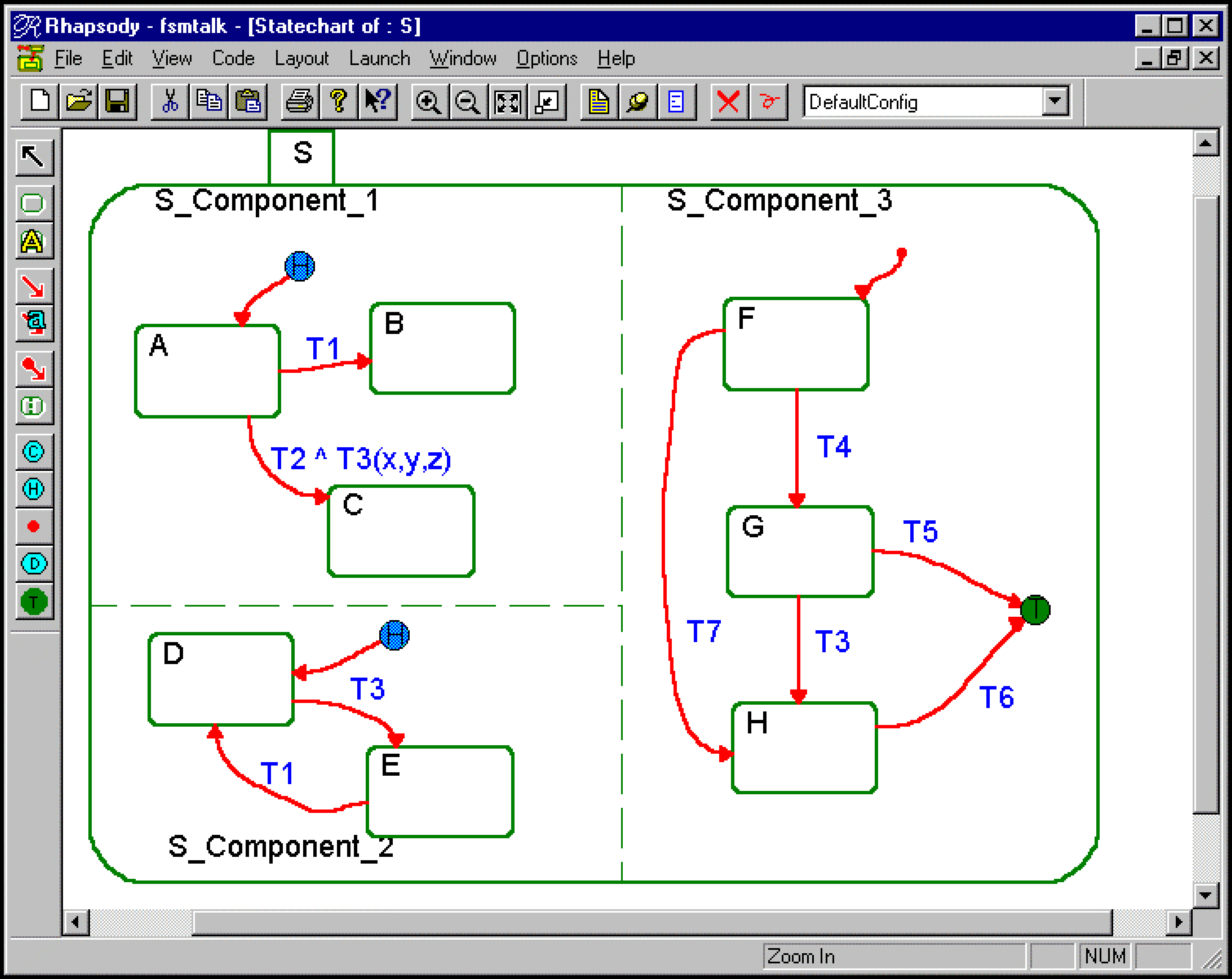
Example Concurrent FSM



Communication in Concurrent FSMs

- Broadcast events
  - Events received by more than one concurrent FSM
  - Results in transitions of the same name in different FSMs
- Propagated transitions
  - Transitions which are generated *as a result* of transitions in other FSMs

Propagation and Broadcasts



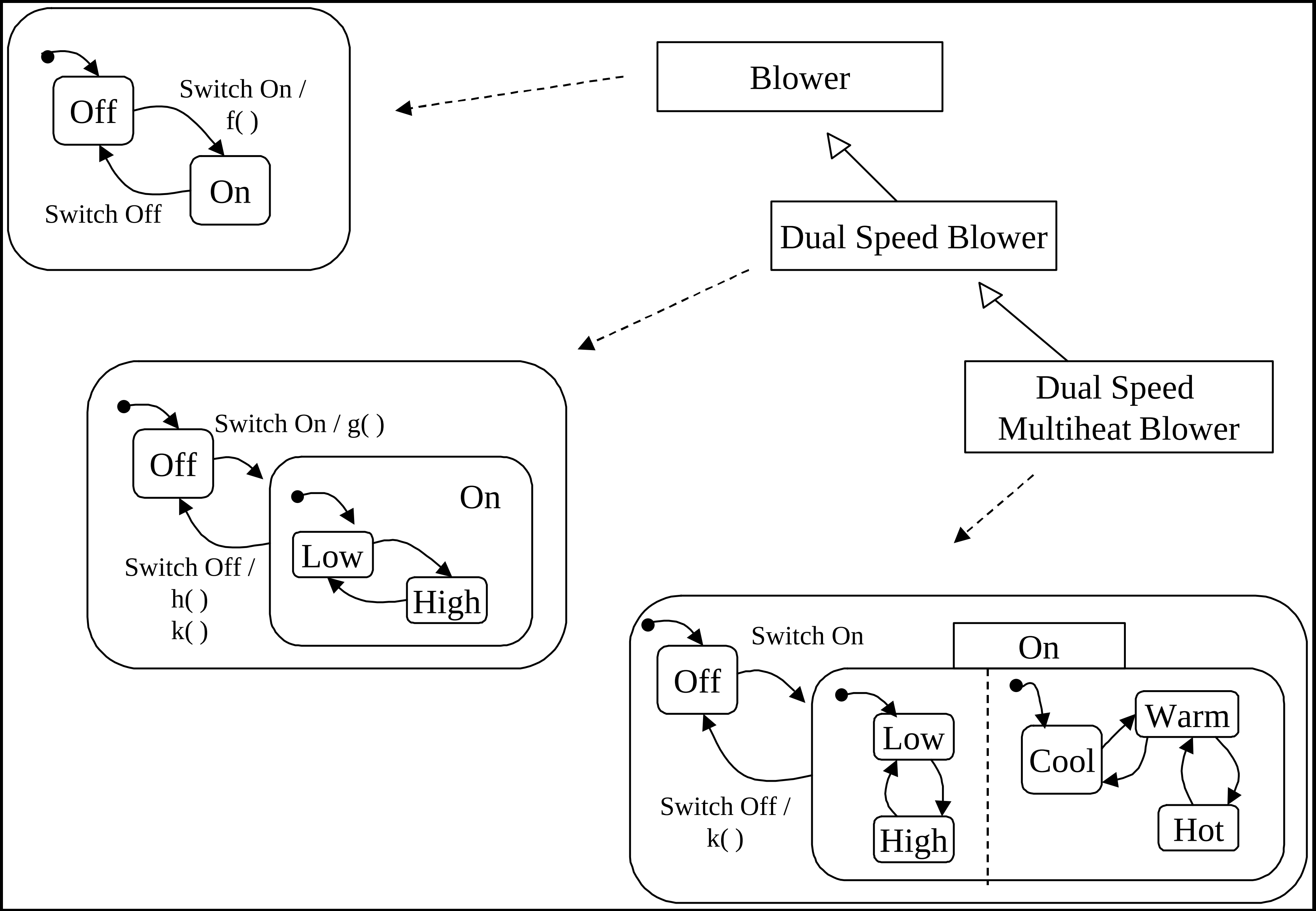
Inherited State Behavior

- Two approaches to inheritance for generalization of reactive classes
  - Reuse (i.e. inherit) statecharts of parent
  - Use custom statecharts for each subclass
- Reuse of statecharts allows
  - specialization of existing behaviors
  - addition of new states and transitions
  - makes automatic code generation possible

Inherited State Behavior

- Assumes Liskov Substitution Principle for generalization:
  A subclass must be freely substitutable for the superclass in any operation
- You CAN
  - Add new states
  - Elaborate substates in inherited states
  - Add new transitions and actions
- You CANNOT
  - Delete inherited transitions or states

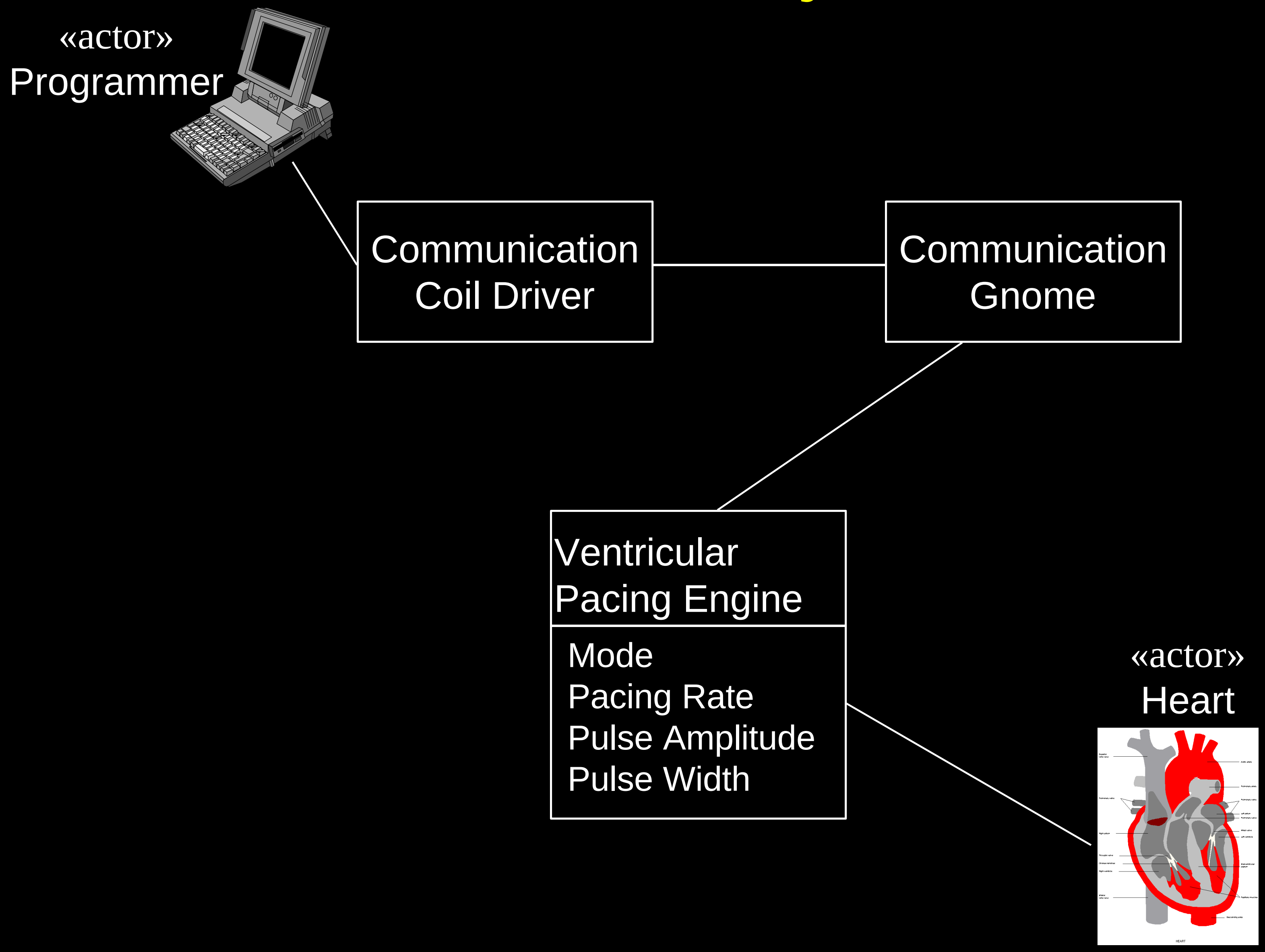
Inherited State Models



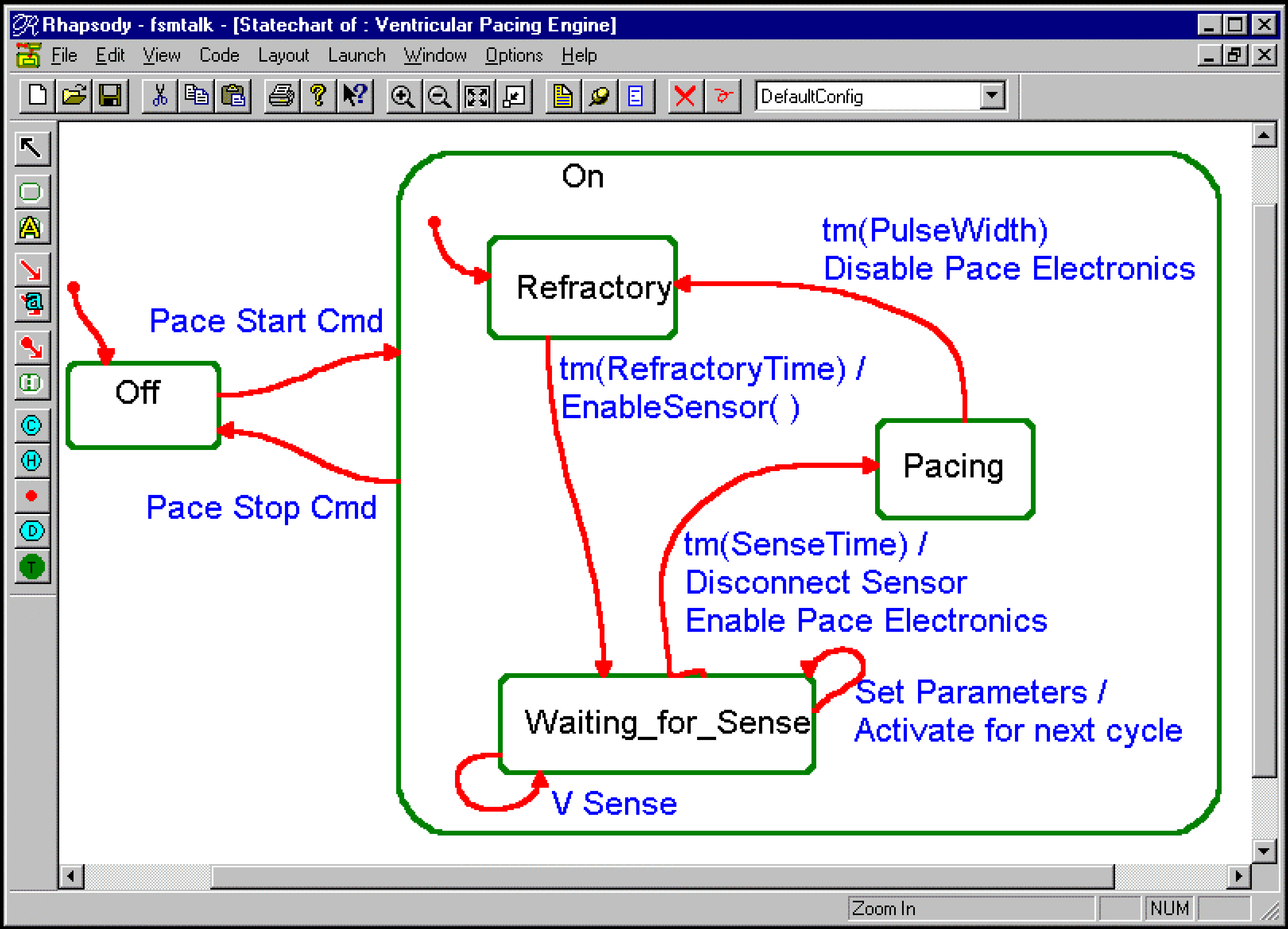
FSM Example: VVI Pacemaker

- 2 key objects executing concurrently
  - Communications object
  - Pacing Engine object
- Each can be modeled as an FSM
- IT IS NOT APPROPRIATE **NOT** TO USE CONCURRENCY IN THIS APP

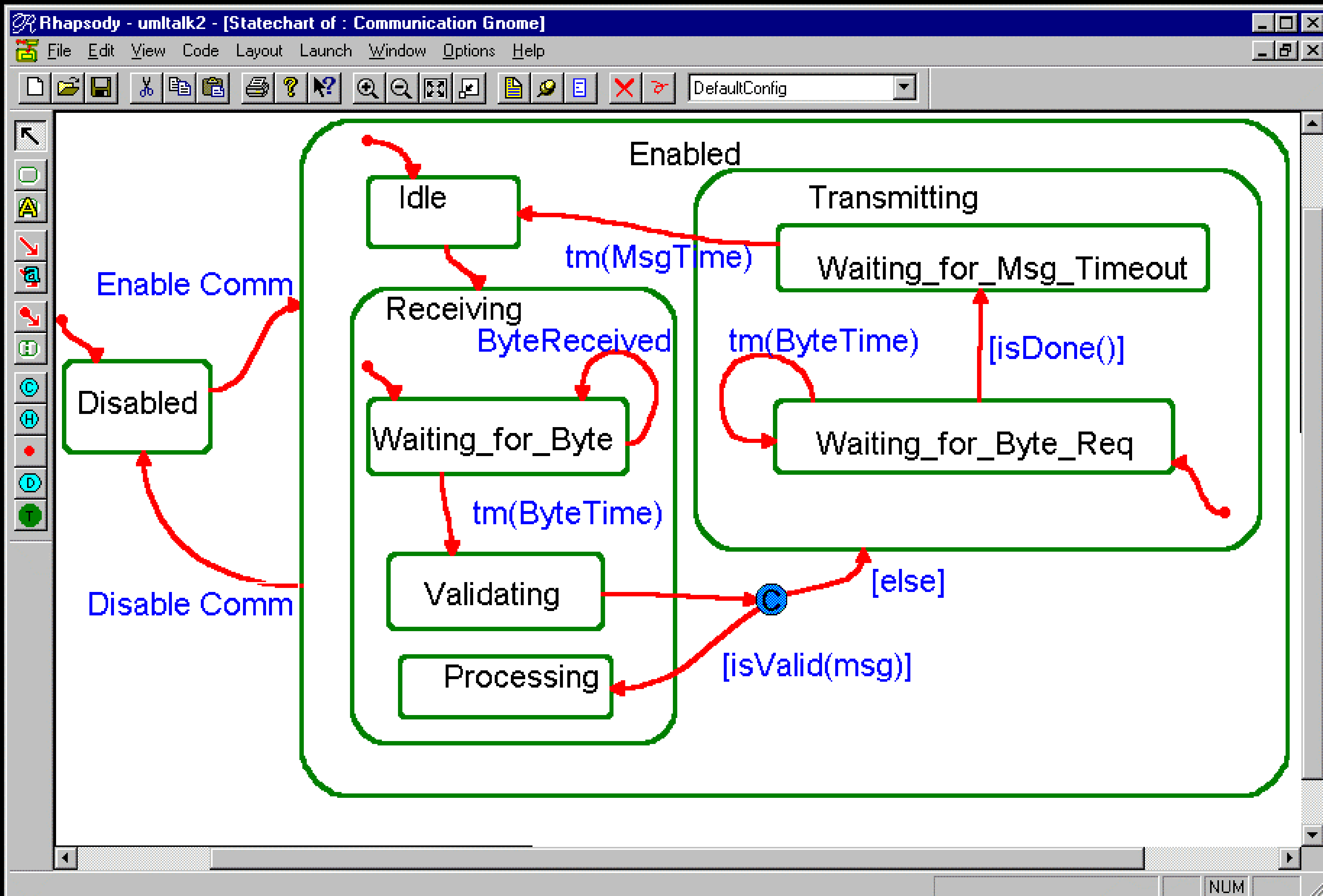
Pacemaker Object Model



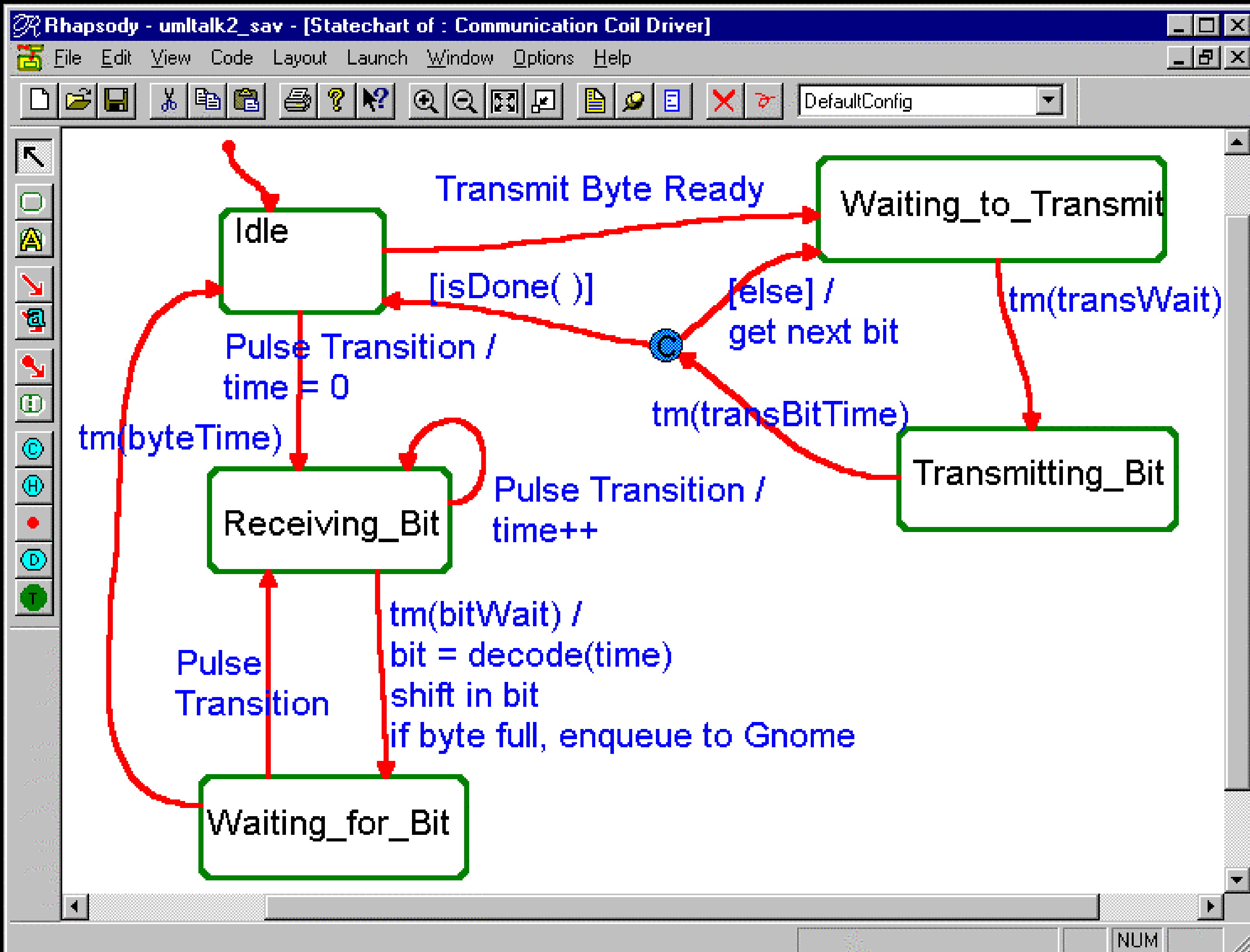
Ventricular Pacing Engine States



Communication Gnome States



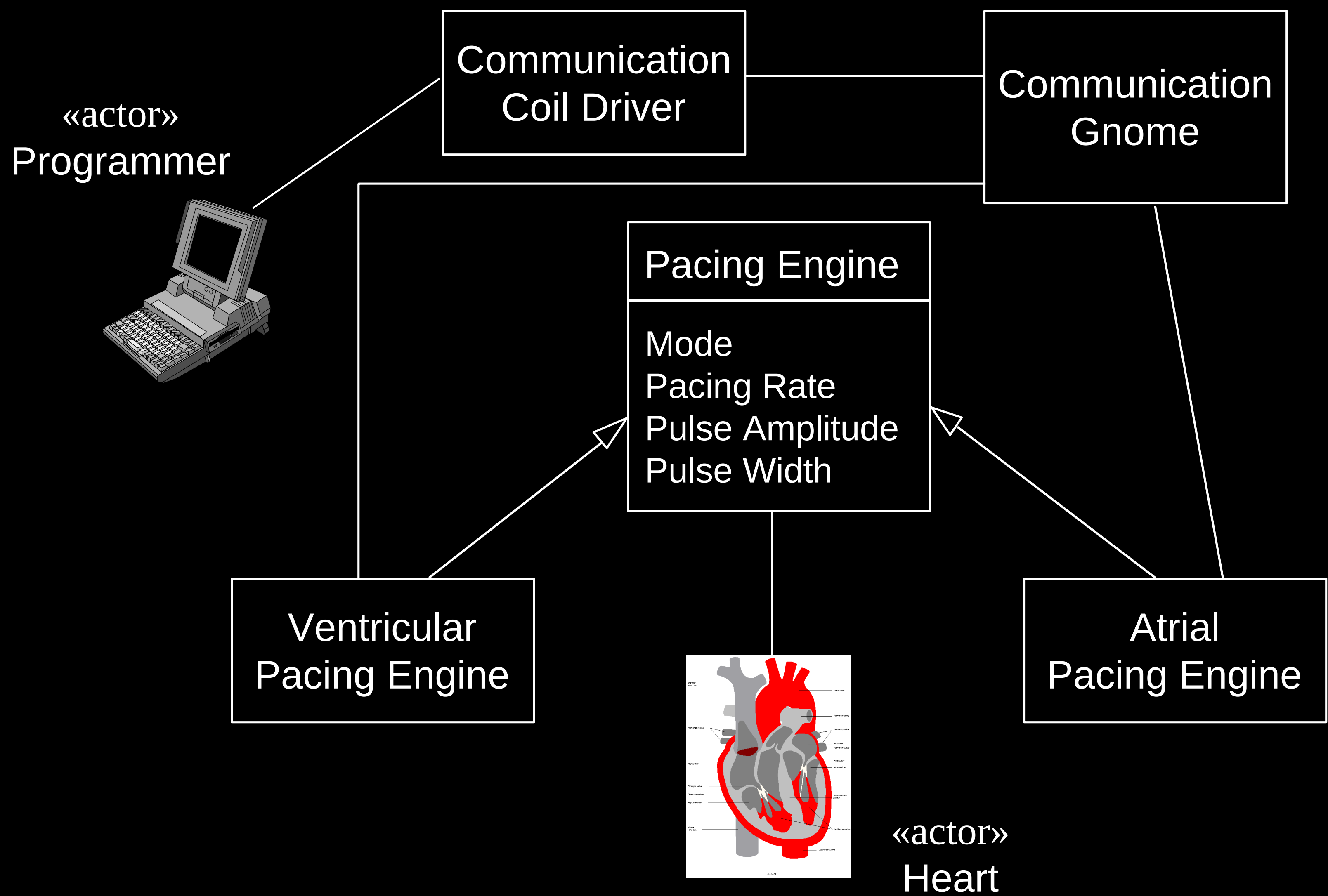
Communication Coil Driver States



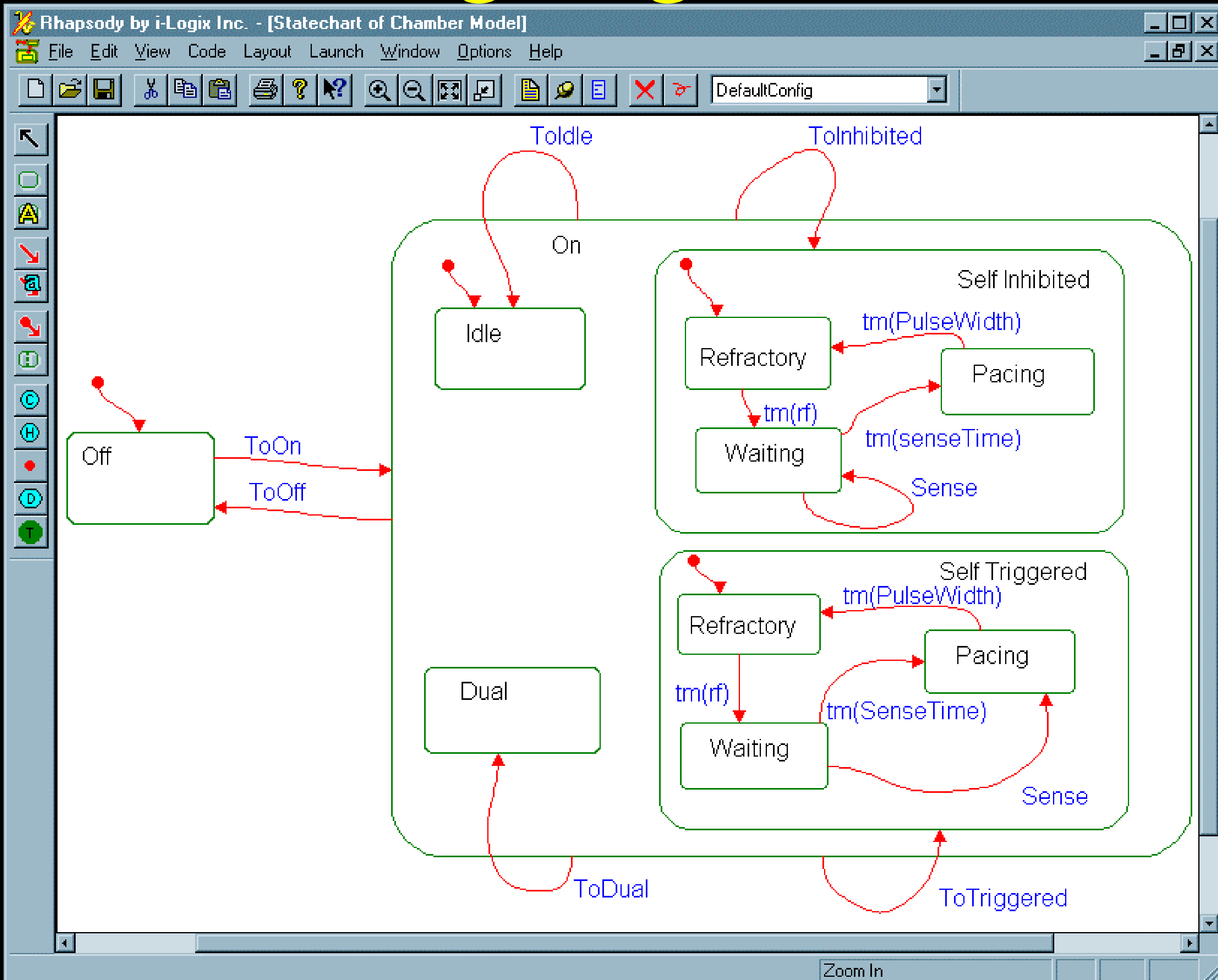
New Pacemaker Spec

- Both Atrial and Ventricular Pacing must be supported:
  - AAI, AAT, VVI, VVT, AVI
- Behavior for AAI is the same as VVI except it is a different object instance
- Behavior for AAT is the same as VVT except it is a different object instance
- Atrial behavior in AVI is different from ventricular behavior

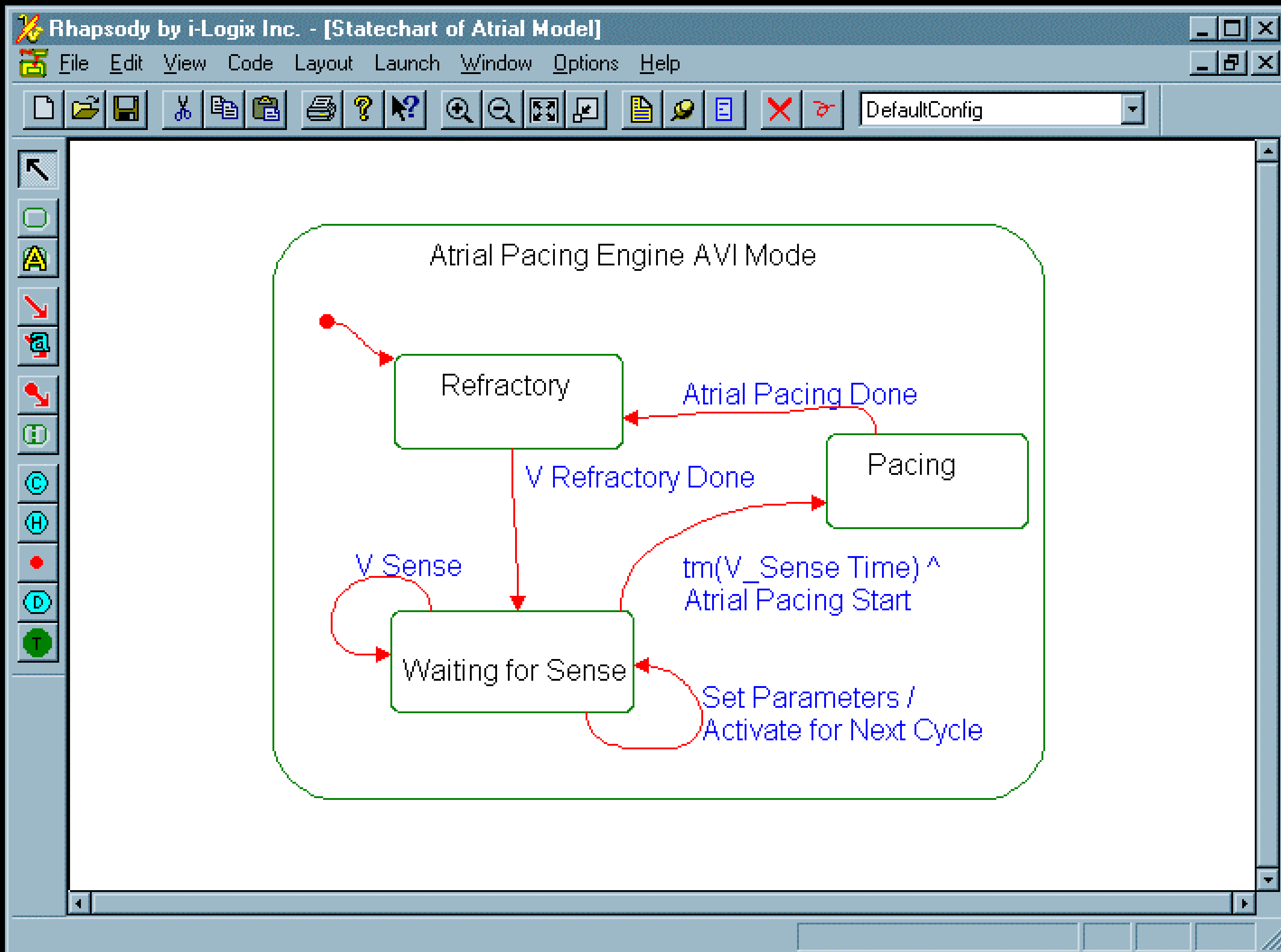
Pacemaker Inherited States



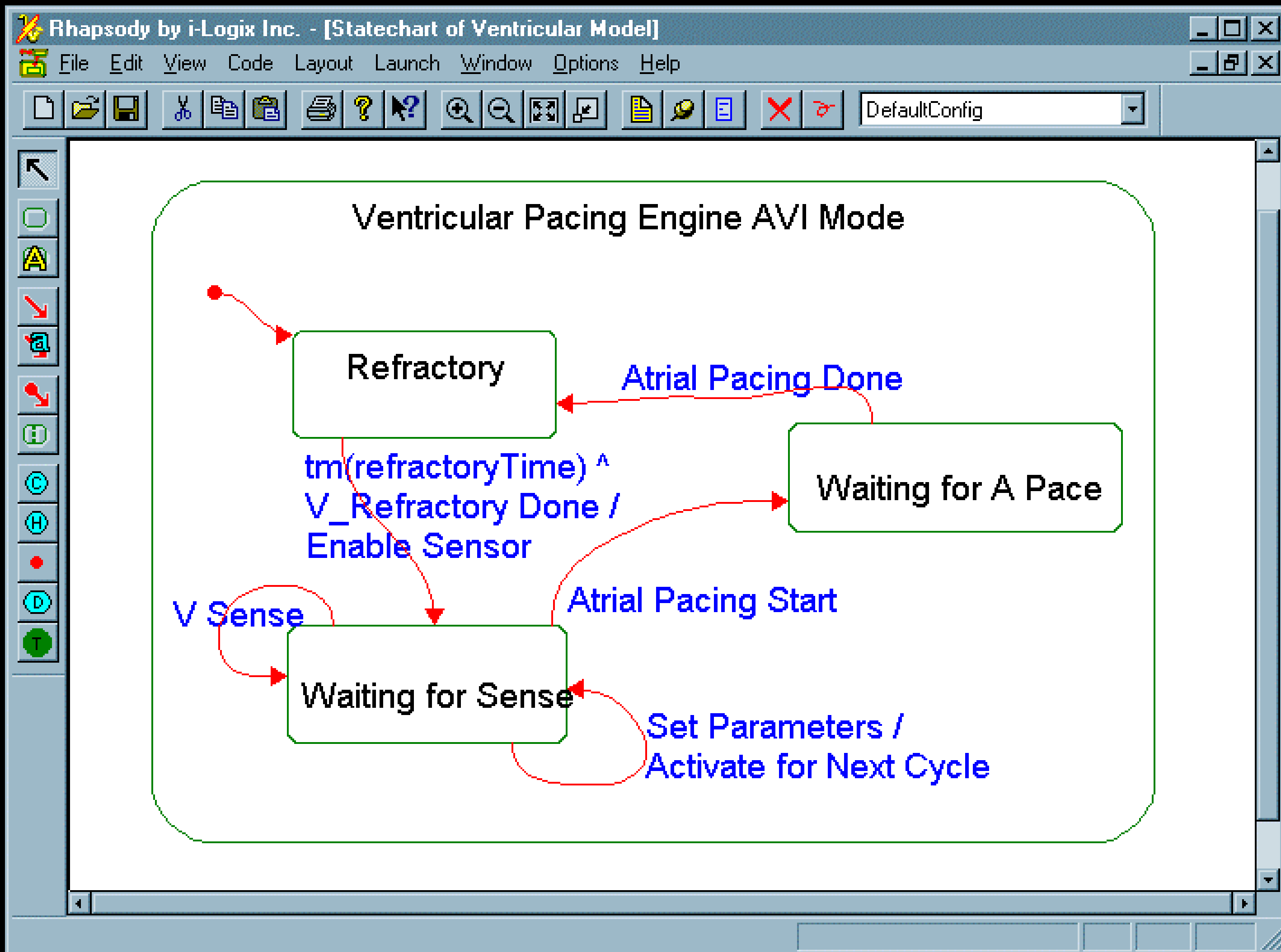
Pacing Engine States



Atrial AVI Mode State



Ventricular AVI Mode State



What is shown in Statecharts?

- Complete state space
- Static structural view
- Supports
  - Nesting
  - Concurrency
  - Propagated transitions
  - Broadcast Transitions

Other State Notations

- State Transition Tables
- State Specifications
- Augmented Message Sequence Diagrams
- Timing Diagrams
- Petri Nets

State Transition Tables

- Arranged as
  - Source *x* Target state
  - Source State *x* Transition
- Statecharts are very good at showing the *structure* of the state space
- Tables are very good at identifying missing transitions
- Shlaer & Mellor say you should do *both*

State Table for VVI Engine

transitions

states

		Stop	Start	Done	Timeout	V Sense	Set Param
1	Off	-	4	-	-	-	-
2	Refractory	1	-	-	4	-	-
3	Pacing	1	-	2	-	-	-
4	Waiting	1	-	-	3	4	4

What's shown in State Tables?

- Complete state space
- Good for seeing missing/erroneous transitions
- No concurrency (one thread per table)
- Propagated transitions
- Broadcast transitions
- No actions

Object (Module) State Specifications

- Work in conjunction with statecharts and state tables
- Textual specifications

State Specifications

- State
  - Name
  - Description
  - Activities
  - Transitions accepted
- Transitions
  - Name
  - Guards
  - Event List
  - Actions List

State Specifications

- Easy to define requirements which are
  - Testable
  - Traceable (good for TUV, FDA, DoD)
- Can fully describe and define the states and transitions
- Recommendation: *Put all three in a single object behavioral document*
  - Statecharts
  - State tables
  - State specifications

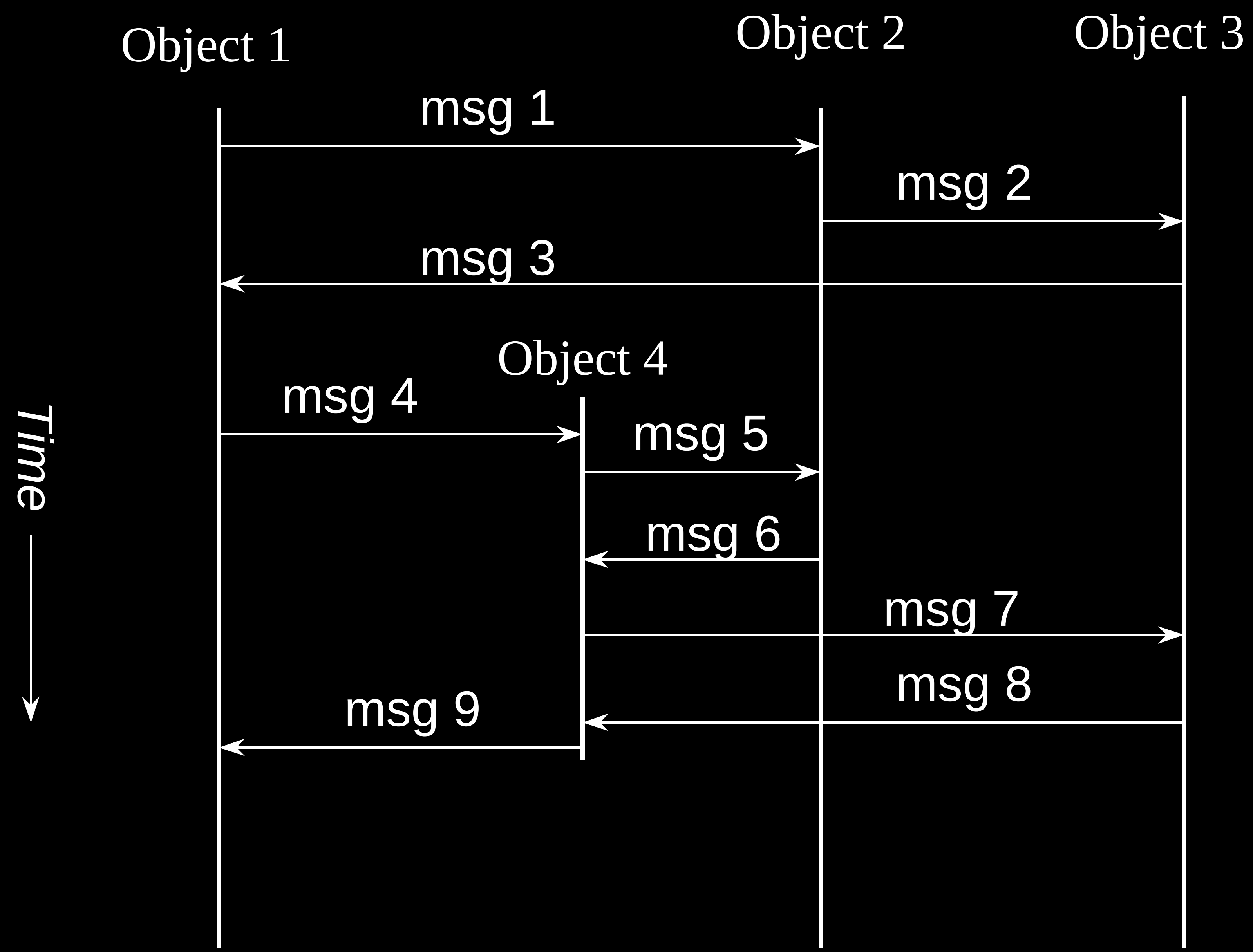
Augmented Sequence Diagrams

- Dynamic
  - Do not show full state space
- Show specific thread through the state space
  - “Scenario”
- Can be augmented with State indicators
- Good for “walking through” behavior
- Do not replace static structural views

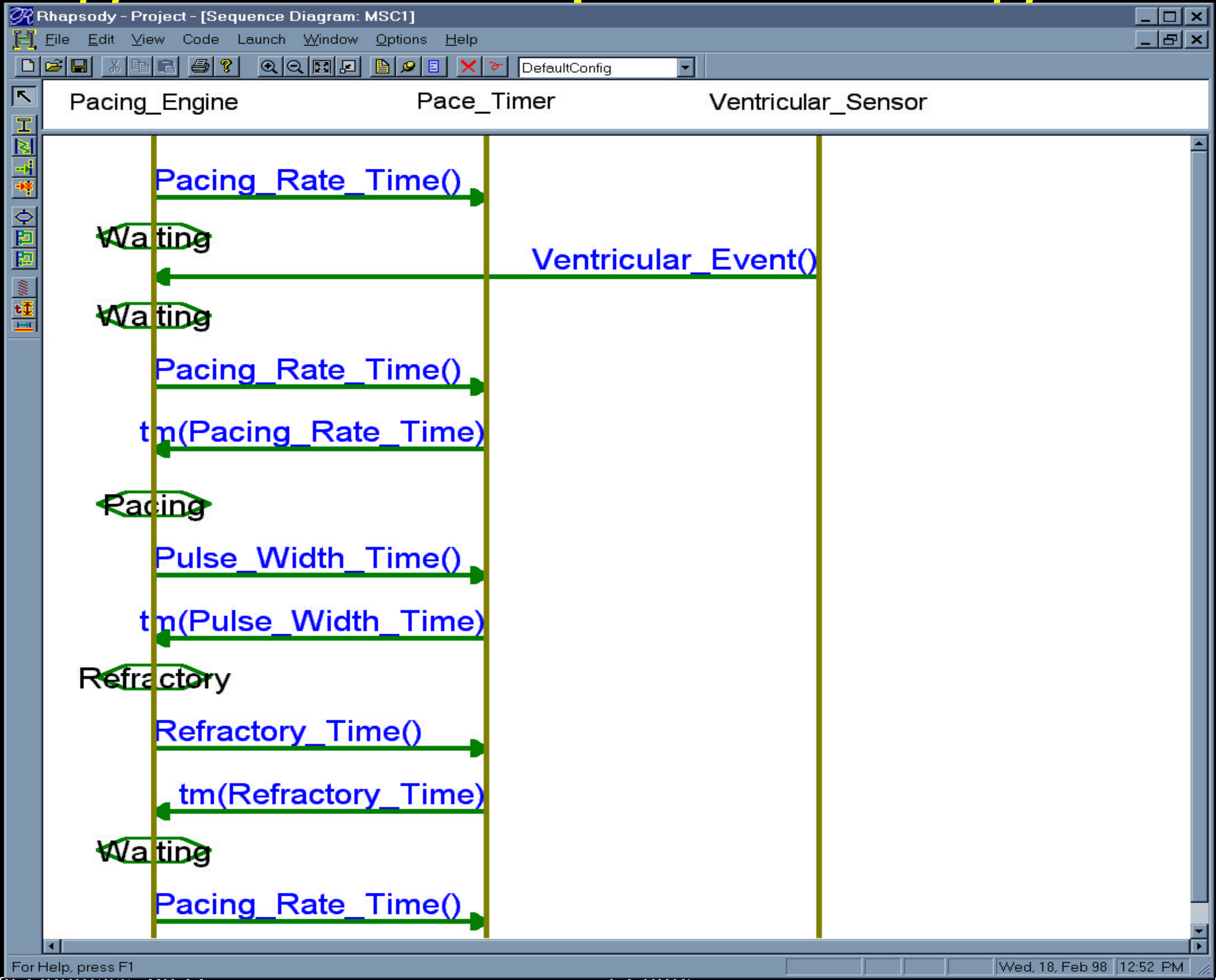
Sequence Diagrams

- Vertical lines represent objects
- Horizontal arrows represent messages (incl. transitions)
- Time flow from the top of the page downwards
- Sequence only is shown normally

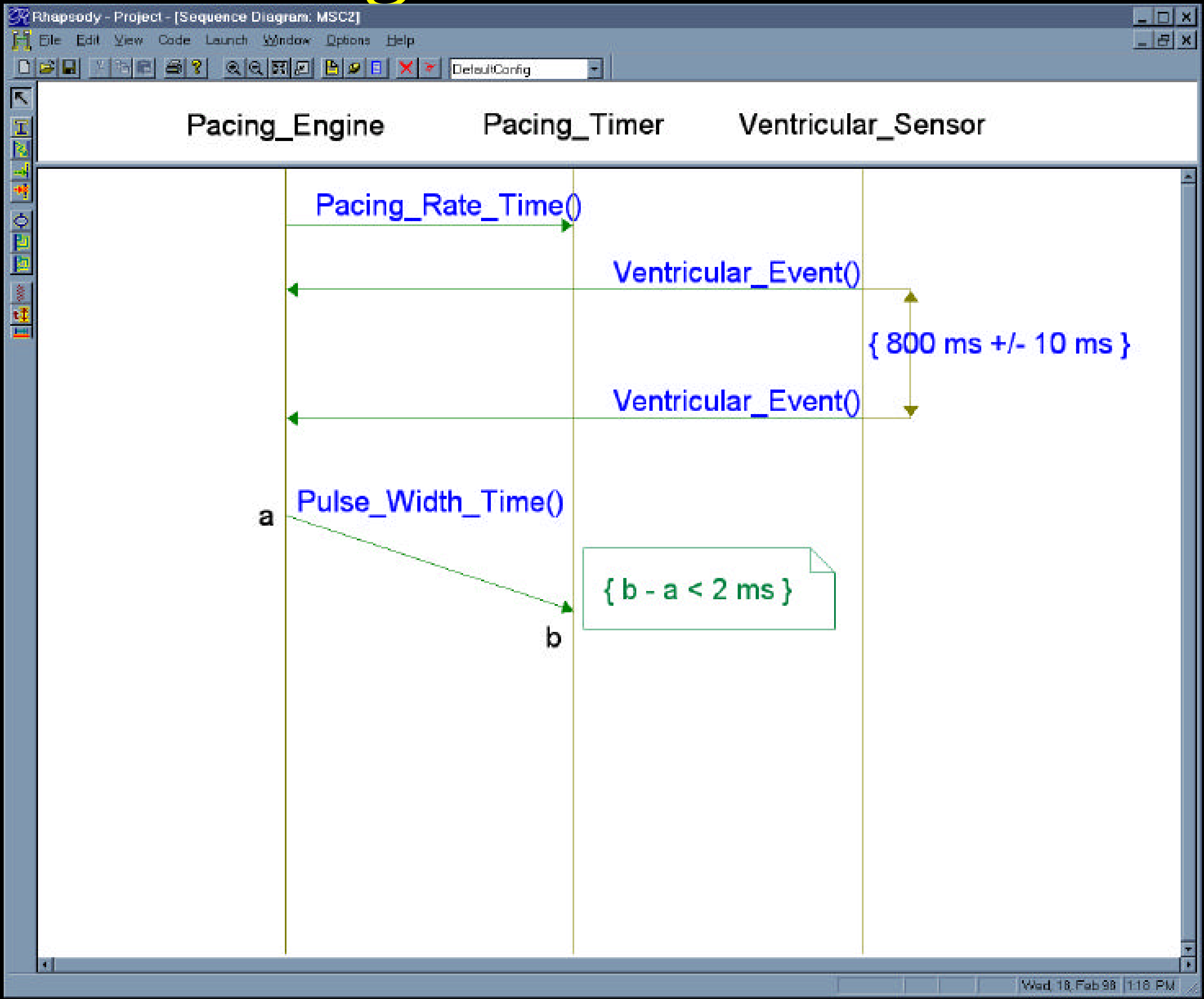
Sequence Diagrams



Augmented Sequence Diagrams



Adding Time Annotations



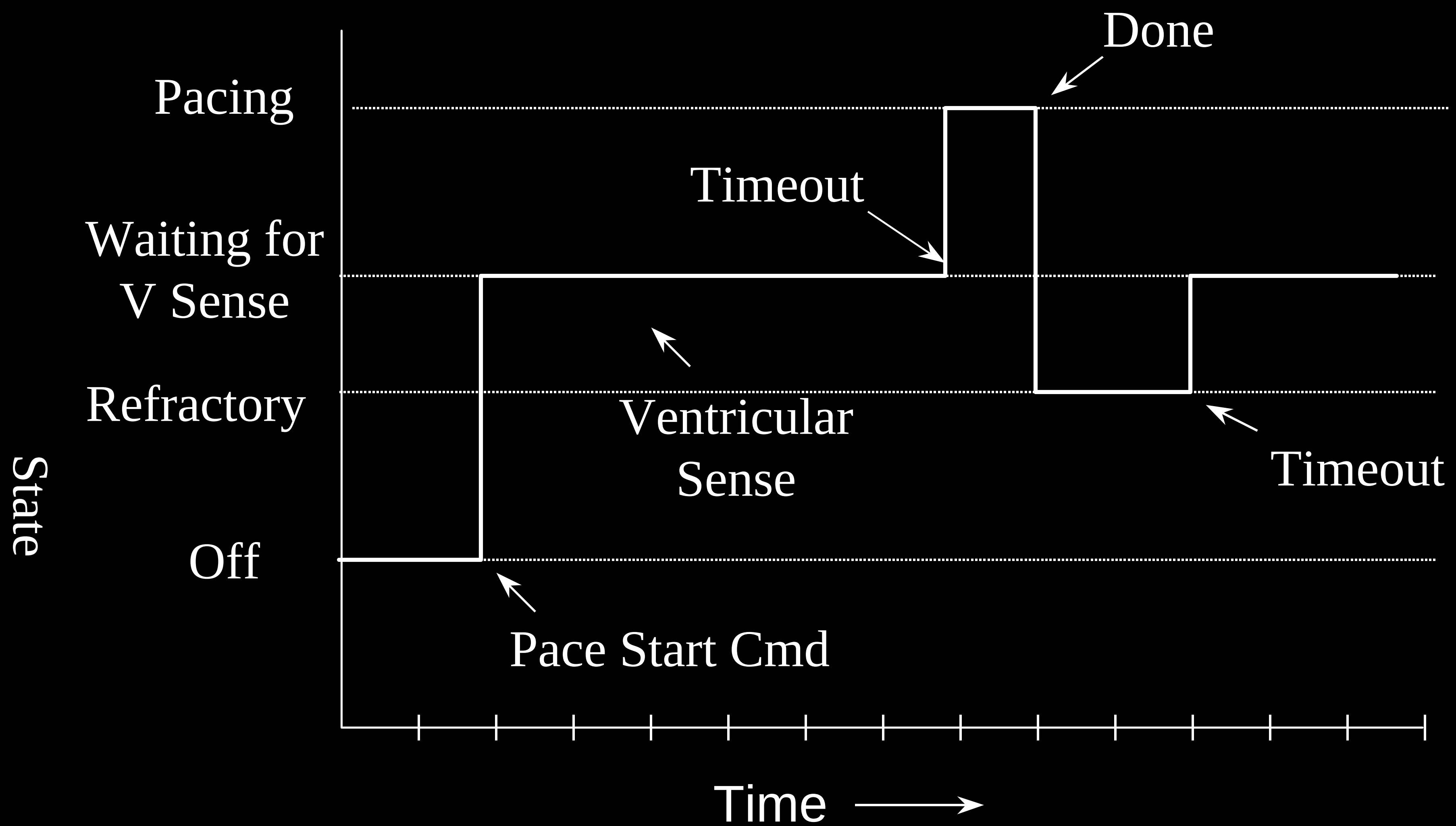
What's shown in Augmented Sequence Diagrams?

- Dynamic scenarios
  - typically a single state chart will result in *many* ASDs
- Good place to add dynamic timing information
- Not all messages result in state transitions

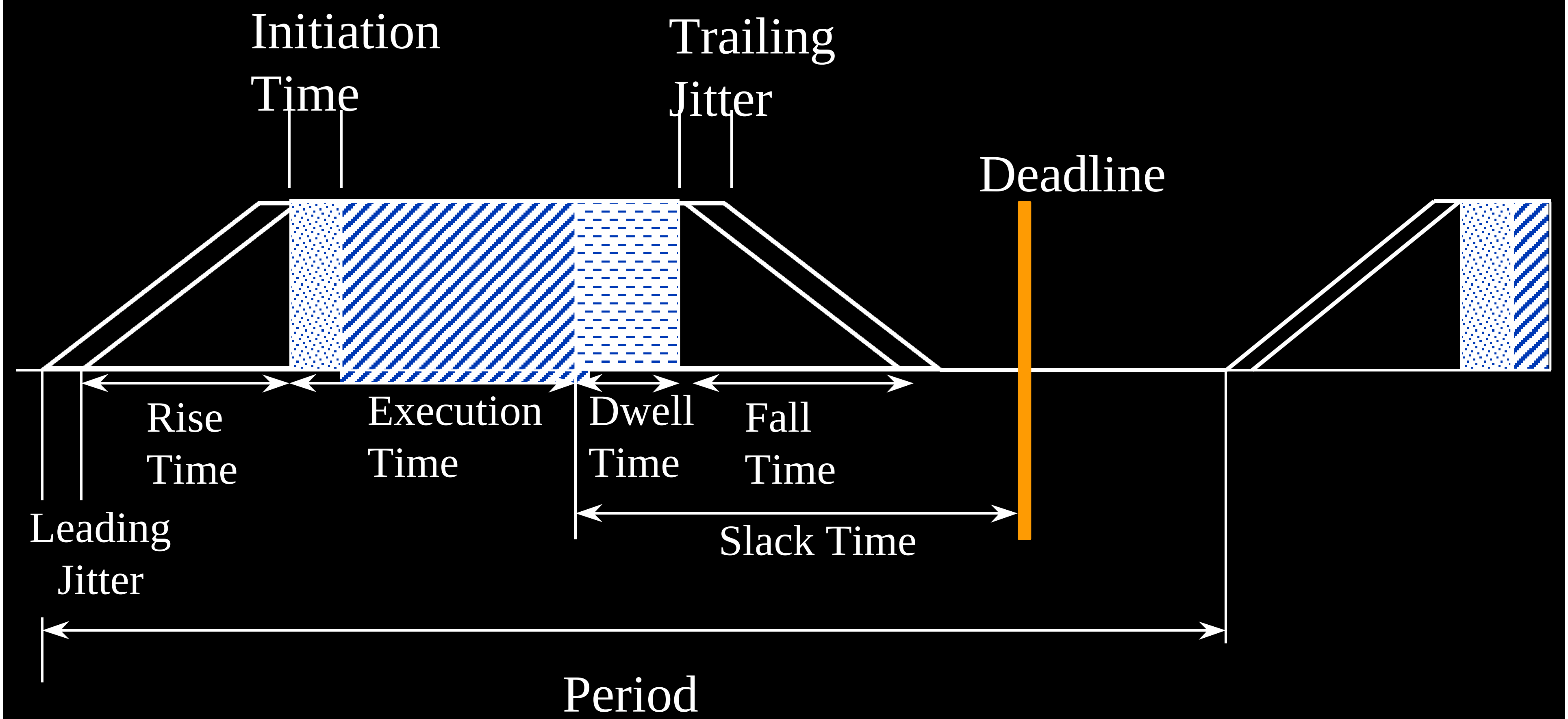
Timing Diagrams

- Familiar
  - Used by electrical engineers
- Show state along vertical axis
- Show linear time along horizontal axis
- Depict particular scenarios
- For usage see
  - *Real-Time UML: Efficient Objects for Embedded Systems* (Addison-Wesley, Oct. 1997)
  - *Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time Applications* (Addison-Wesley, Spring 1999)

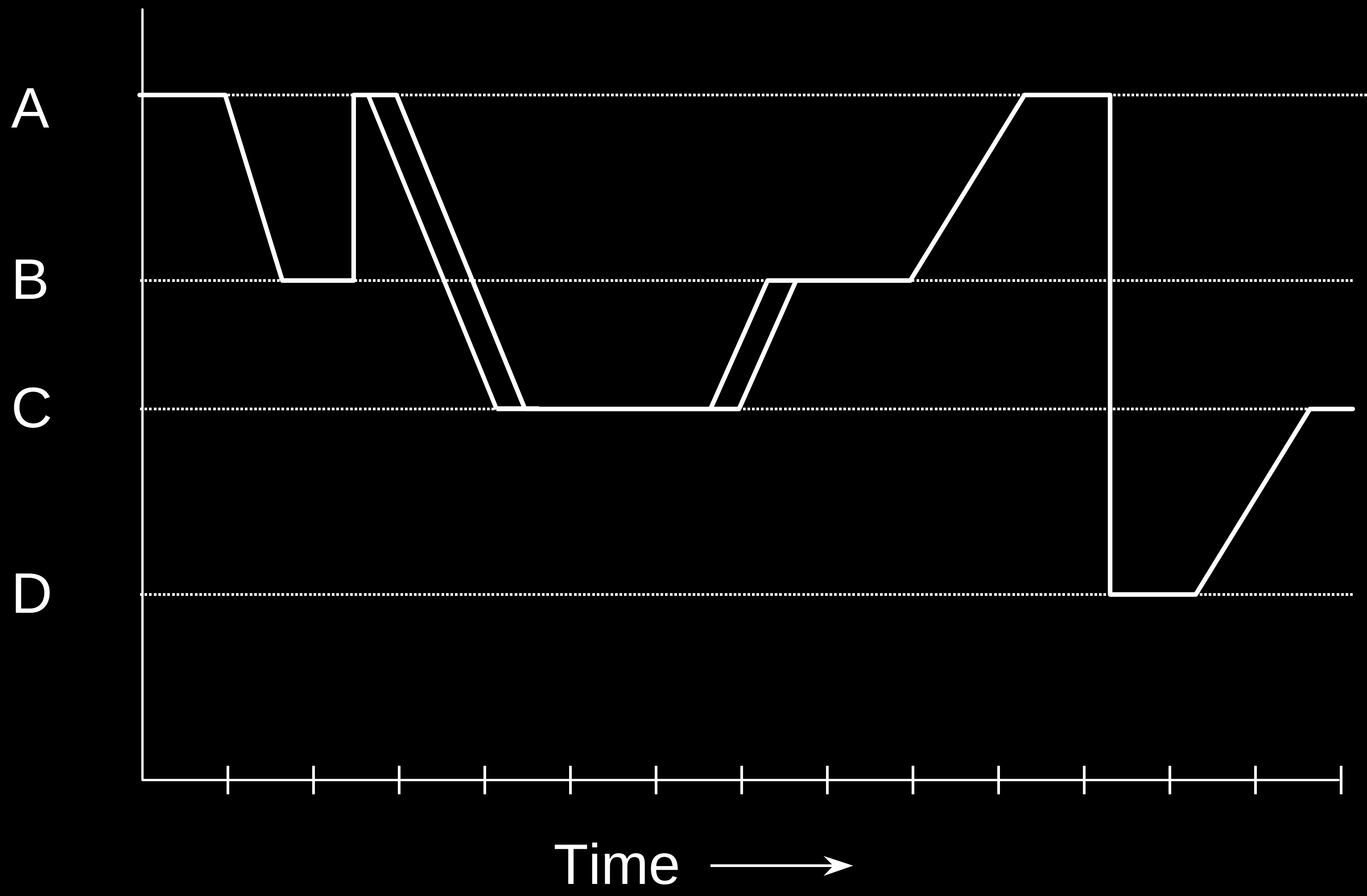
Simple Timing Diagram



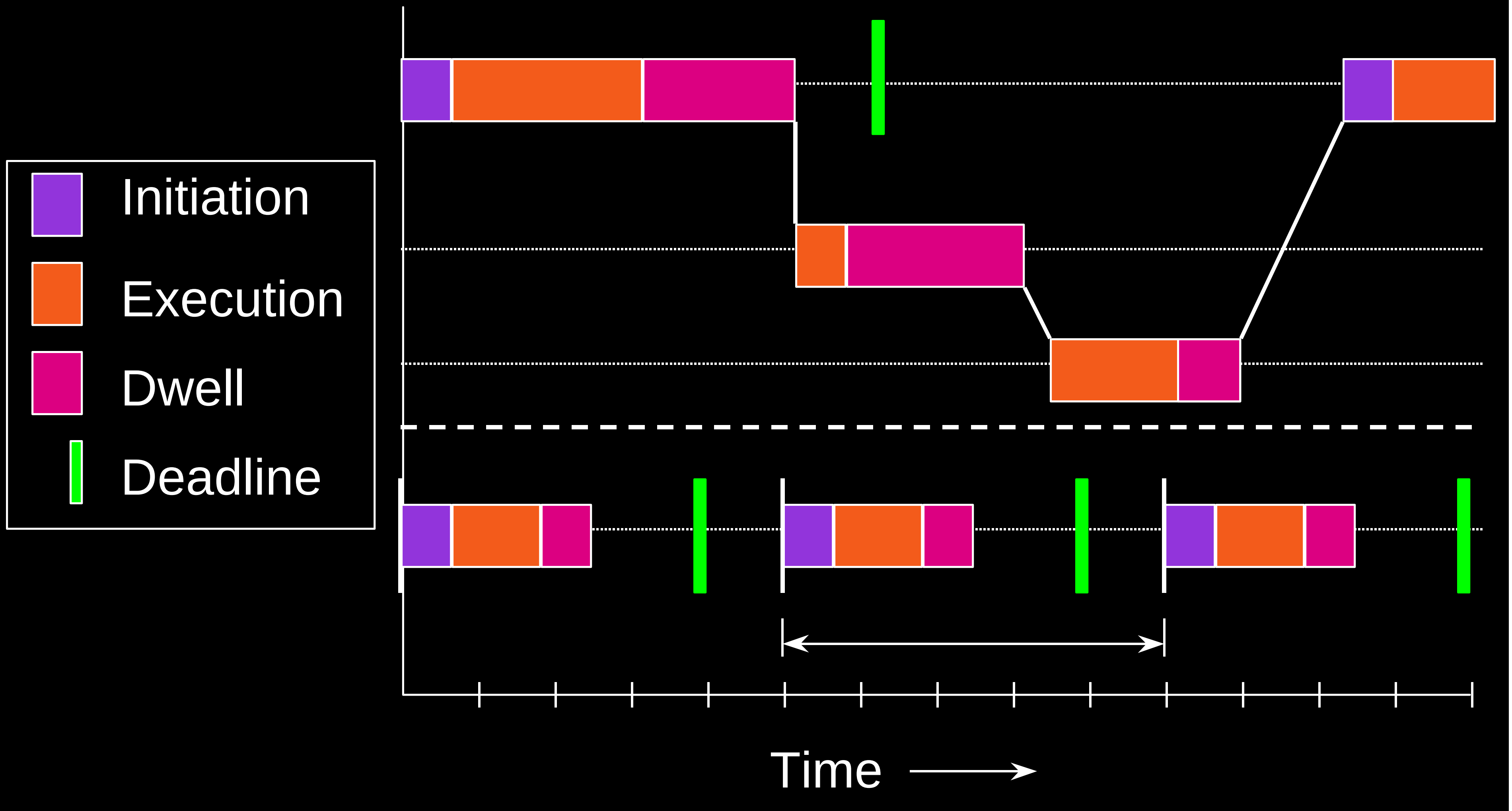
Complex Timing Diagram



Example with jitter and rise times



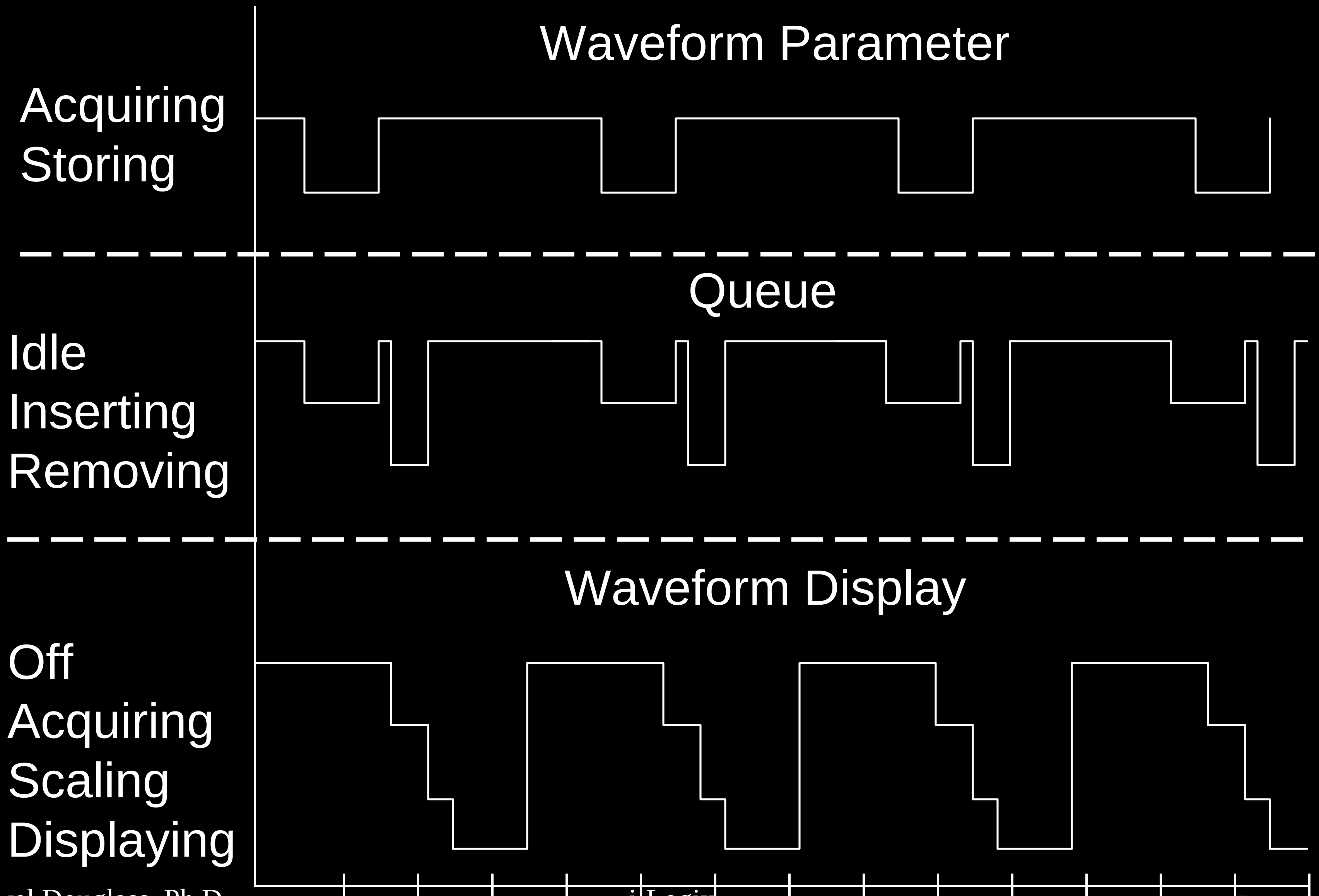
Example with Dwell and Slack



Concurrency in Timing Diagrams

- Concurrency can be shown by creating horizontal “bands” of states
  - Usually one band per object
- Shows the timing relationships between concurrent threads

Concurrency in Timing Diagrams



Other Applications of Timing Diagrams

- Show timing relationships of functional call threads
- Show testable time budgets
- Assist in understanding RMA results
- Shows sequence of states and object reactions to events

What's shown in Timing Diagrams?

- Good view of overall time
- Timing of interaction of concurrent states
- Timing details
  - Jitter
  - Execution time
  - Dwell time
  - Slack time
  - Rise and Fall time

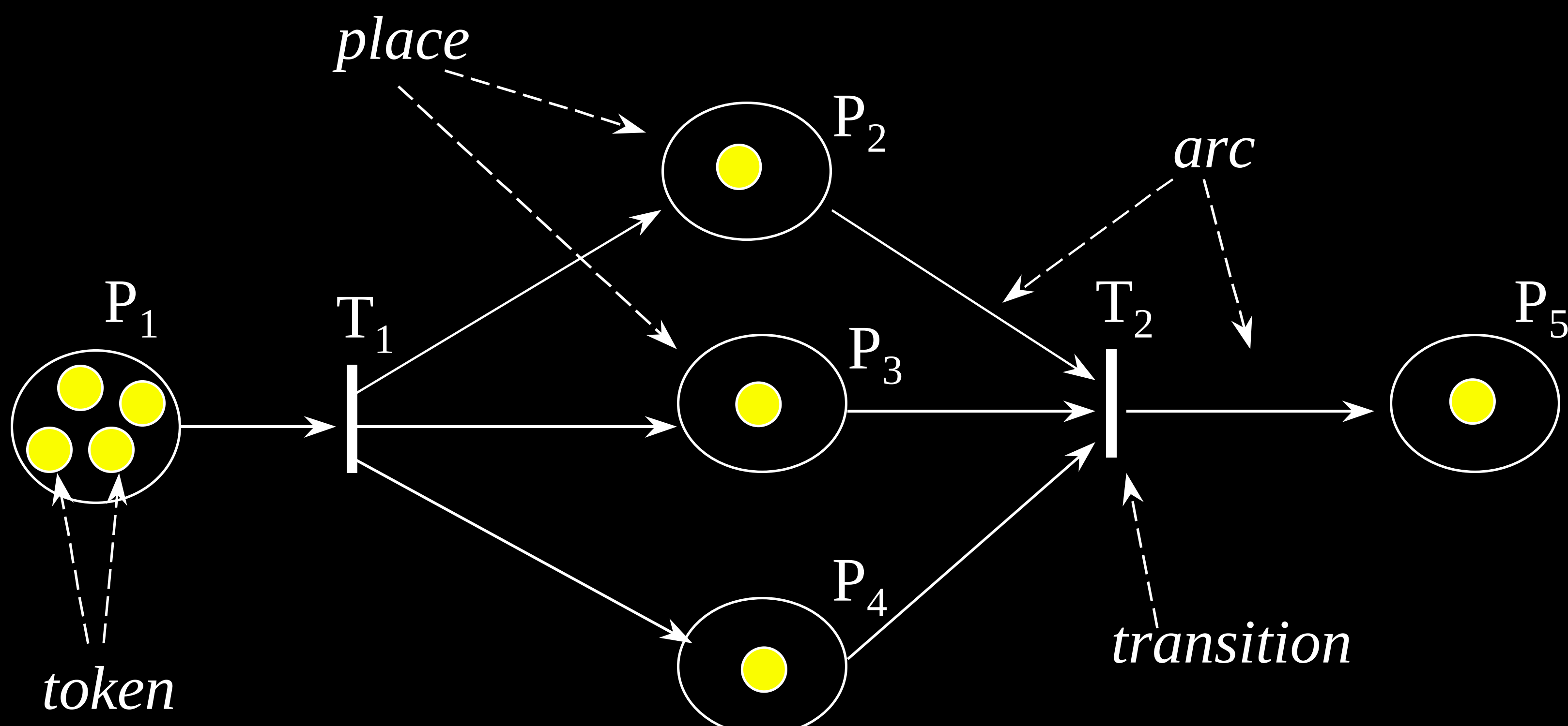
Petri Nets

- Petri nets are a generic modeling tool
- FSMs are a special case of Petri nets
- Petri nets are defined as a set of
  - Places which hold tokens
  - Tokens small filled circles
  - Arcs directed lines
  - Transitions bars connecting arcs from places to places
- Petri nets can show concurrency by permitting multiple tokens

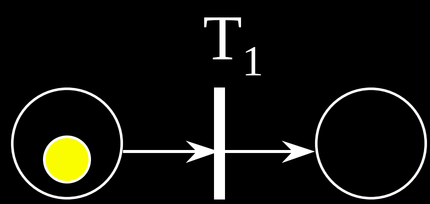
Petri Net Rules

- A Petri net is executed by moving tokens
- A transition can fire iff all of its input places contain tokens
- The firing of a transition
  - Removes a token from each input place
  - Puts a token in each output place
- The number of tokens a place can hold is called its *capacity*

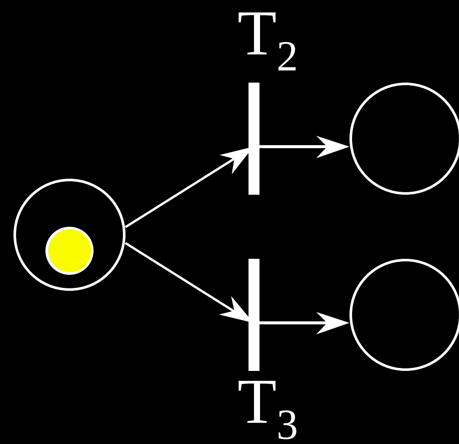
Simple Petri Net



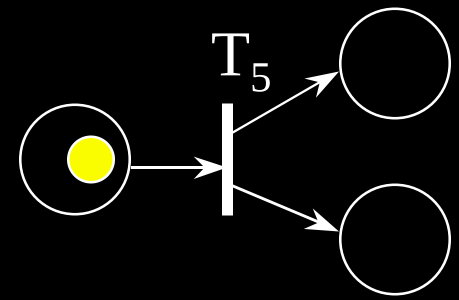
Standard Programming Constructs



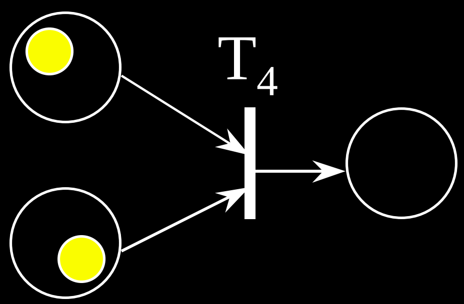
Sequencing



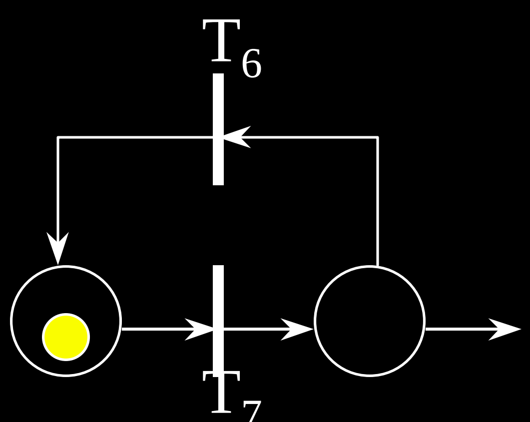
Selection
(or contention)



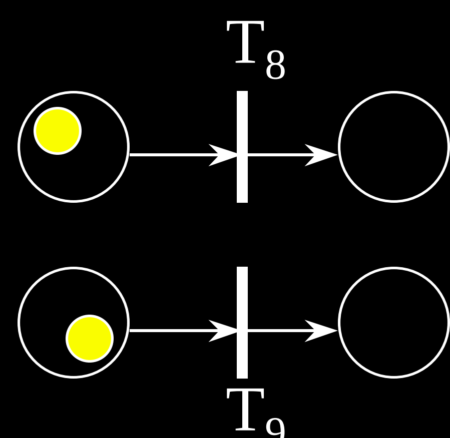
Explicit Control
Branching



Explicit Control
Synchronization

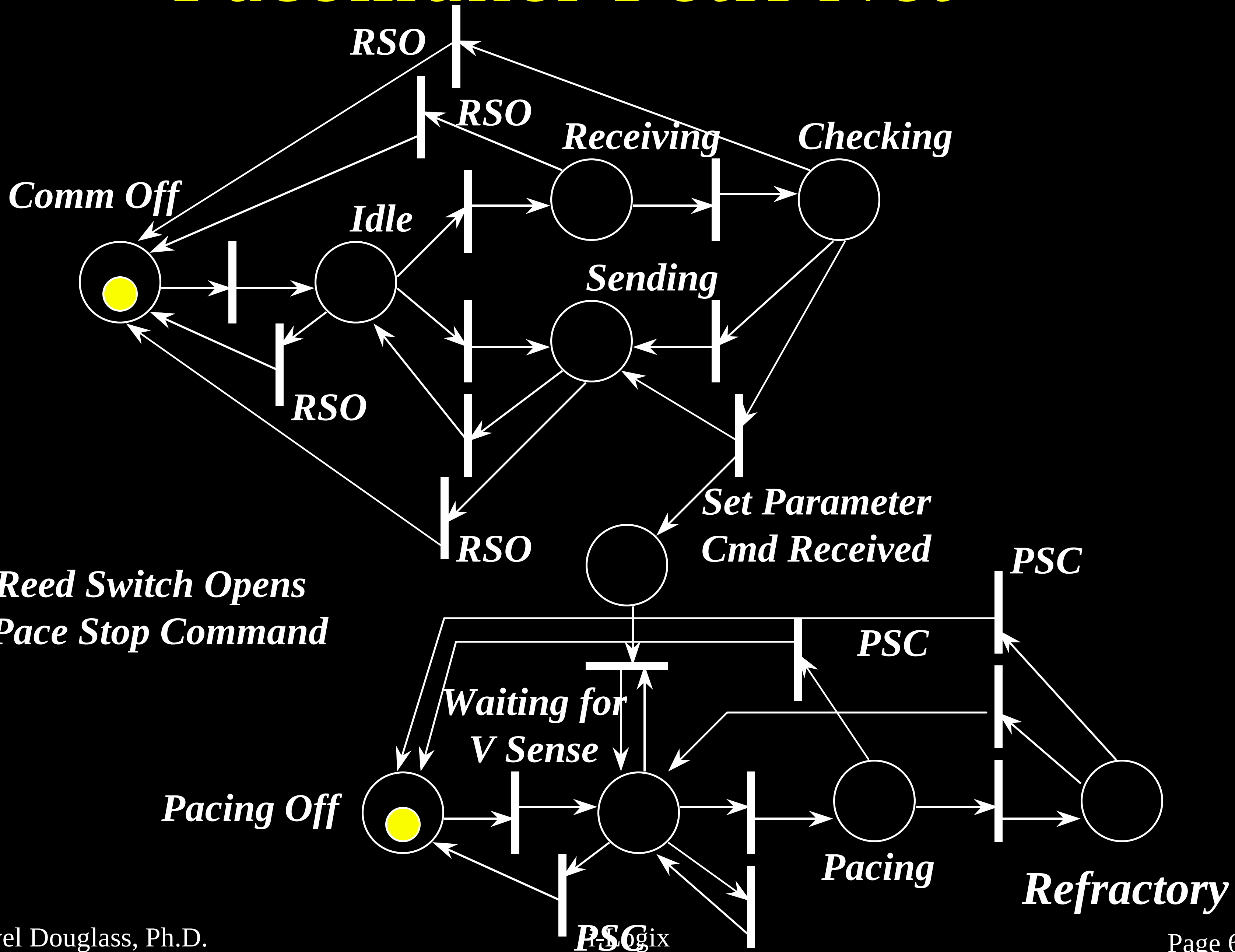


Looping



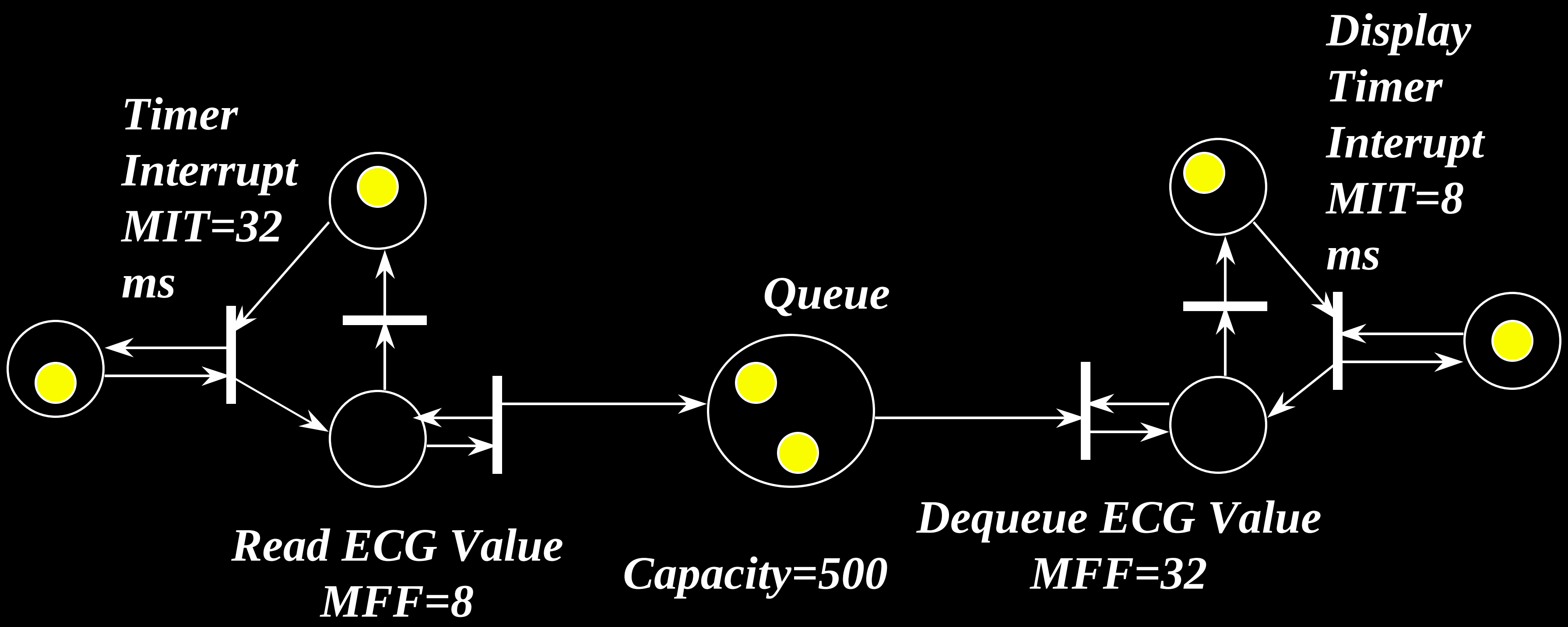
Concurrent threads

Pacemaker Petri Net



RSO **_Reed Switch Opens_**
PSC **_Pace Stop Command_**

Time-Augmented Petri Nets



Example above shows a queuing model between two asynchronous threads: ECG Waveform acquisition and display

What’s Shown in Petri Nets?

- Generalized behavior (incl. state behavior)
- Concurrency
- Can be augmented with time
- Many different extensions are available
- Petri nets suffer from
  - lack of scalability because they are flat like Mealy-Moore state models
  - lack of tools

FSMs and Development Process

- FSMs apply to *OBJECTS*
  - Sensor object
  - Queue object
  - Pacemaker pacing engine object
  - Language parser object

Structured Process

- Identify behavioral functions that exhibit state behavior
- For each such function, design a FSM
  - For each state, define
    - Valid transitions
    - Actions
    - Activities
- Decide on an implementation strategy

Object Oriented Process

- Identify classes and objects
- Identify which classes have FSMs
- Define a single FSM for each relevant class
  - For each state, define
    - Valid transitions
    - Actions
    - Activities
- Decide on an implementation strategy

Implementation Strategies

- Case/Switch statements
- FSM Generator
- Centralized state machine
- Separate state machines for each FSM object

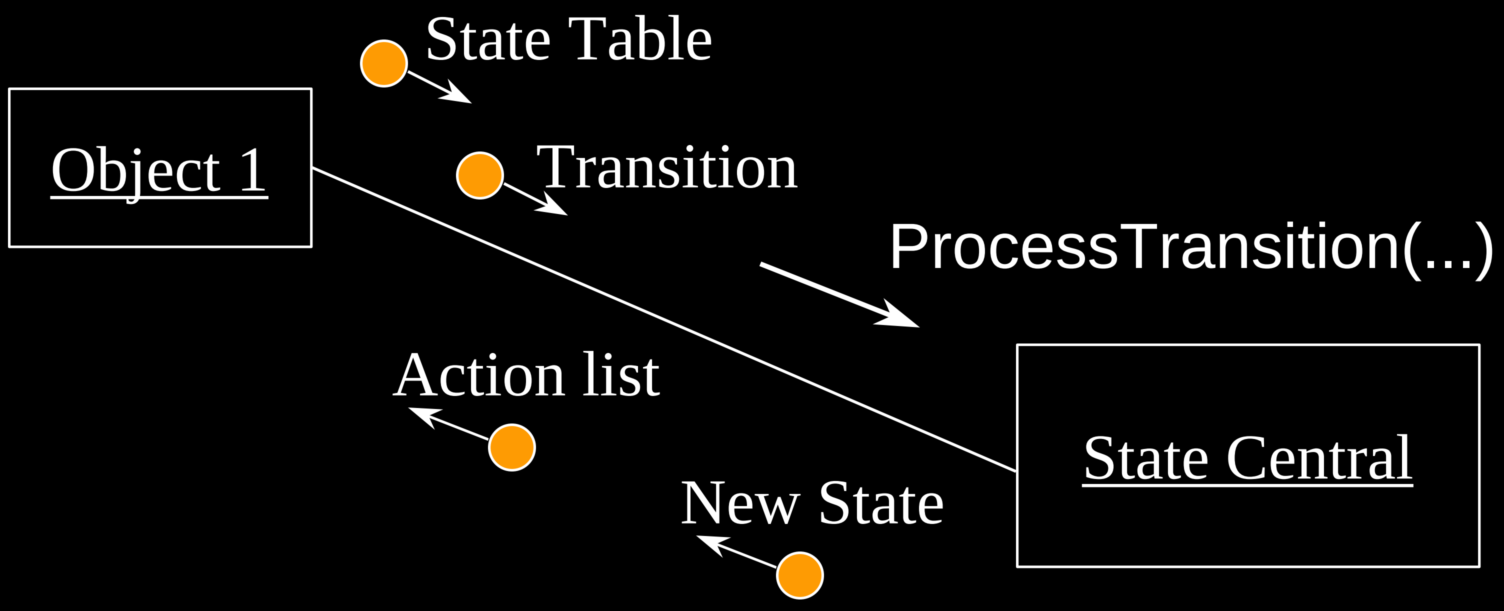
Case/Switch Statements

```
Switch (stateVar) {
      case state1:      ...
                        break;
      case state2:      ...
                        break;
      case state3:      ...
                        break;
      case default:     // invalid state
      };
```

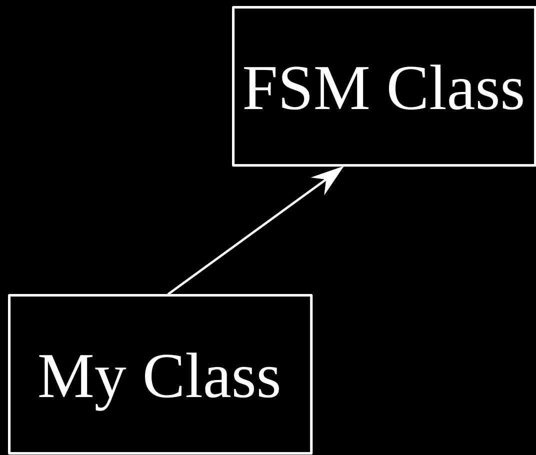
Case/Switch Statements

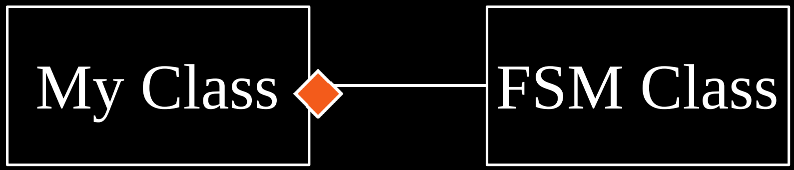
```
Switch (stateVar) {
    case state1:    switch(transition) {
                    case T1:    ...
                                break;
                    case T2:    ...
                                break;
                    case default:   // invalid transition
                    };
                    break;
```

Centralized State machine



Separate State machines



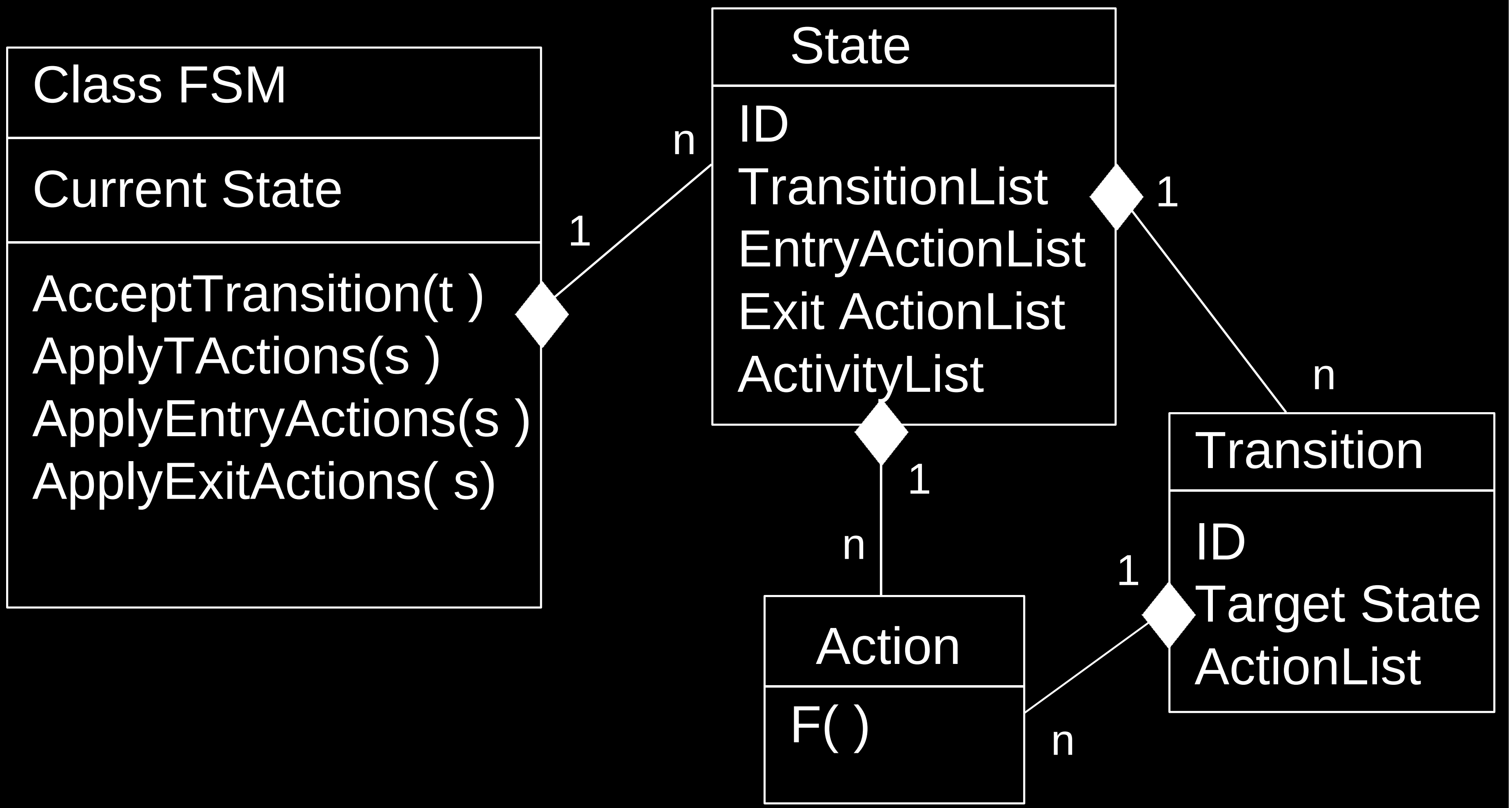



```
class myClass: public FSM
{

};
```

```
class myClass {
private:
        FSM myFSM;
};
```

Separate State Machines



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 (which may contain concurrent 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 provide
  - Nested States
  - Concurrency
  - Propagated and Broadcast Transitions
  - Orthogonal Components
  - Guards on transitions
  - Flexible action model
  - Activities within states
  - History
  - Inherited state behavior

Summary

- Statecharts show static structural view
- State tables show missing transitions
- State specifications are good for defining testable, traceable requirements
- Sequence diagrams show scenarios
- Timing diagrams show overall timing in scenarios
- Petri nets are more general and show static structural view


i-Logix
Embedded Systems and Software Design Automation