CIS 721 - Real-Time Systems
Lecture 29: Design Modeling

Mitch Neilsen
neilsen@ksu.edu

Outline

- Embedded System Design
  - Requirement Modeling – functional requirements (Use Cases)
  - Analysis Modeling
    - Structural Object Analysis – static model defining the relationships between classes (Class Diagrams, etc.)
    - Behavioral Object Analysis – model describing dynamic (behavioral) aspects (statecharts, etc.)
  - **Design Modeling – design a software architecture**
    - **Architectural Design – system-wide**
    - **Mechanistic Design – inter-object**
    - **Detailed Design – intra-object**
  - **IBM Rational Rhapsody**

IBM Rational Rhapsody

- Includes a graphic editor for each of the possible UML diagrams
  - Use case, sequence, collaboration, object model, component, state machine and activity diagrams

- Not only capture the design of the system but also generate implementation code
  - C, C++, Java and Ada

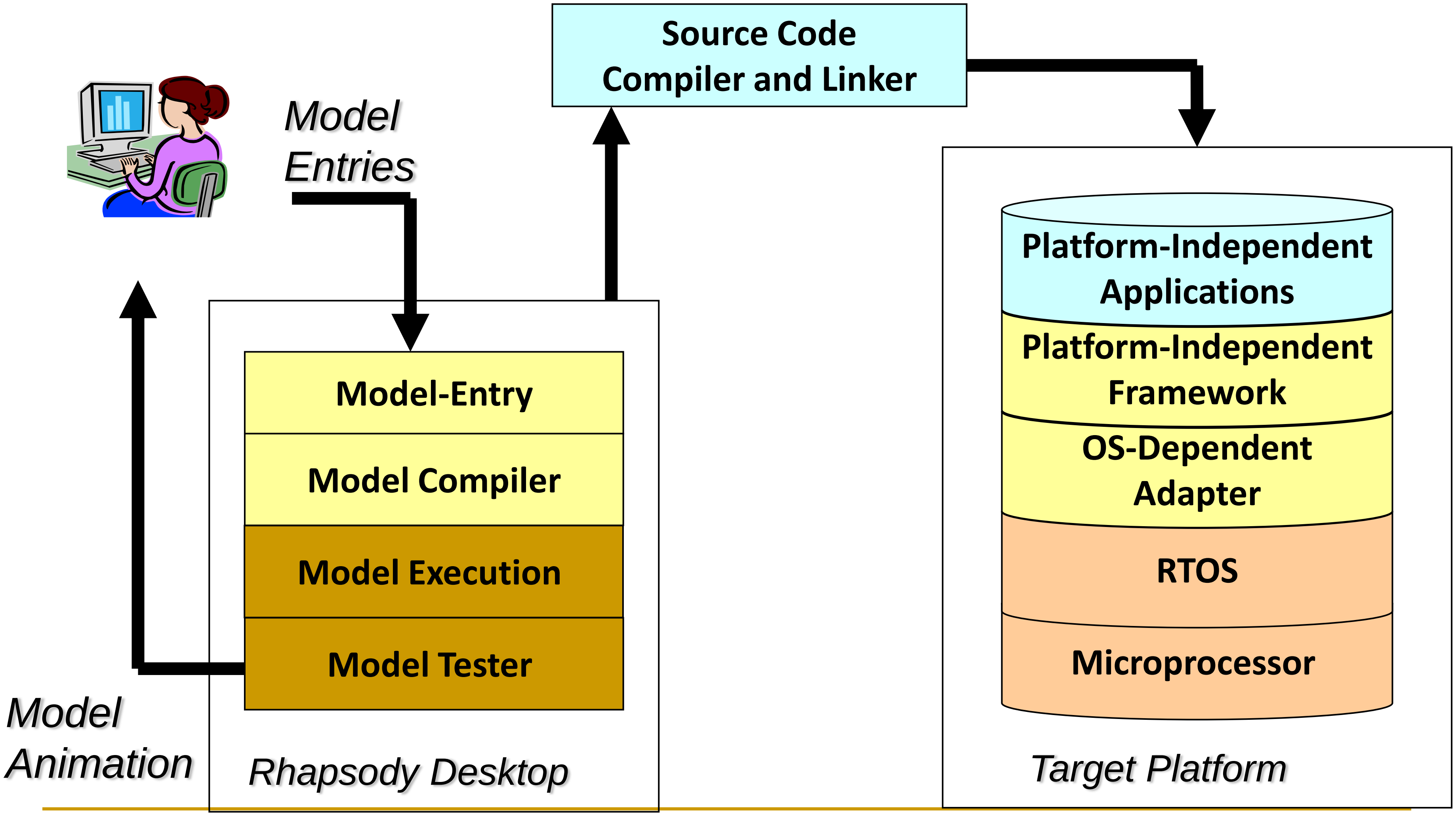
Rhapsody for RTOS

- Provides Model-Driven Development (MDD) environment based on UML 2.1 – 2.5
  - Systems and software development of real-time and embedded applications
- Can use an iterative design approach
  - Software can be constantly executed and validated
- Can rapidly target the platform independent application model to a real time embedded operating system

Rhapsody for RTOS

- Can construct portable and technology independent systems via
  - Generation of application code from platform independent models (PIMs)
  - Object eXecution Framework (OXF)
  - Use of OS-specific adaptors for most commercial RTOSs

Rhapsody for RTOS



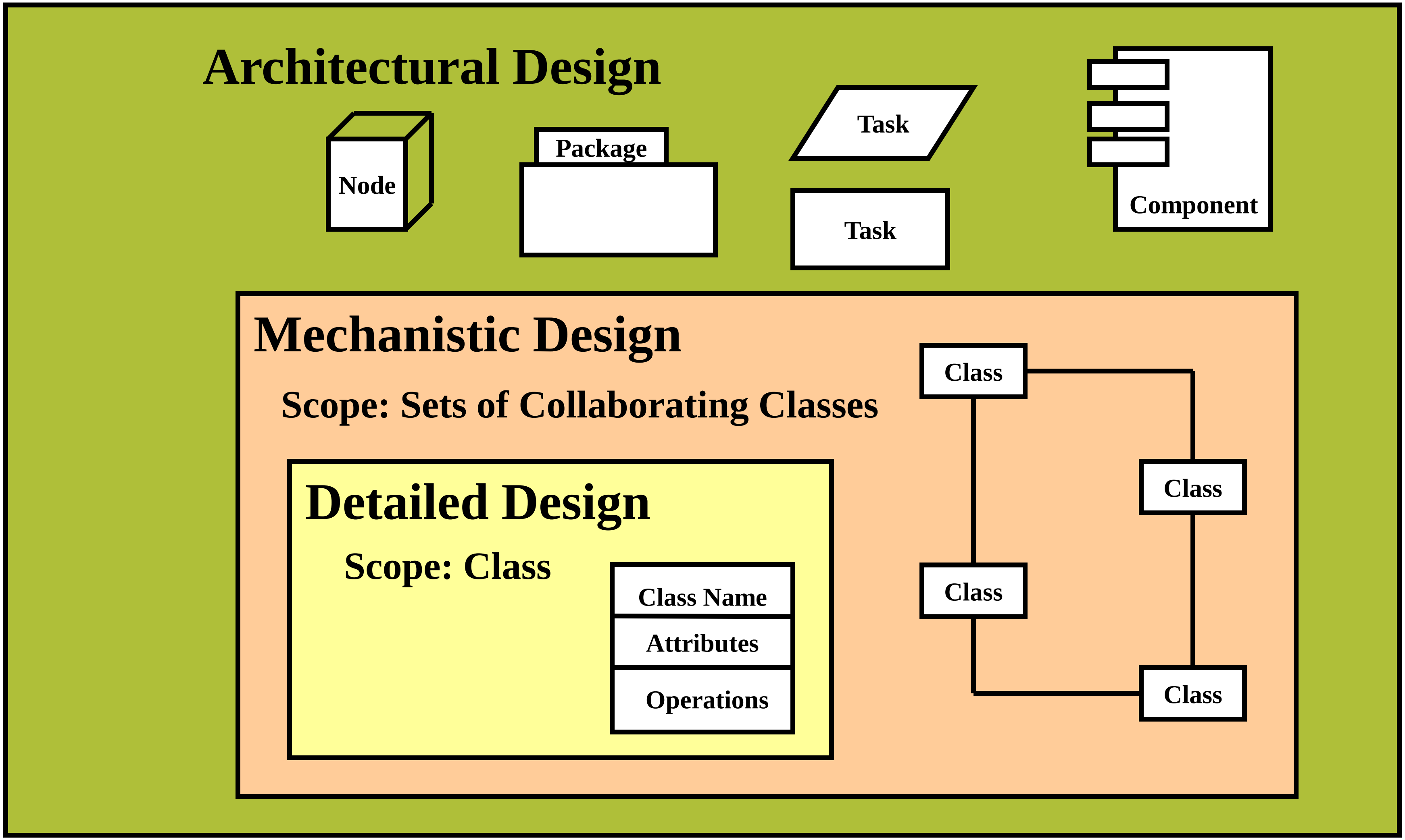
Stereotypes

- What is a stereotype in UML?
- A **stereotype** (denoted << >>) is a language extension mechanism that is used to classify model elements or introduce new types of model elements (called metamodel elements) [UML, 1.4].
  - Stereotypes can be used to extend the UML notational elements; e.g., to classify and extend associations, inheritance relationships, classes, and components.
  - **Example:** A **<<capsule>> is a stereotype of a class.**   A capsule is the fundamental modeling element of Rational Rose Real-Time (RoseRT).

Design Modeling

- Specify a solution that is consistent with the Analysis Model.
- Design Categories:
  - **Architectural Design - system-wide**
  - **Mechanistic Design - inter-object**
  - **Detailed Design - intra-object**

Design Categories



Architectural Design Models

- **Tasking Model** - concurrent set of tasks and their interactions
- **Component Model** - run-time artifacts and their interfaces
- **Deployment Model** - mapping of these components to the physical hardware
- **Safety/Reliability Model** - redundant components to provide safety/reliability

Tasking (Concurrency) Model

- Real-time systems typically have multiple threads that execute concurrently.
- A **thread** can be defined as a set of actions that execute sequentially.
- A **task** is a thread and the object(s) in which the thread executes.

Active Classes

- Class and object diagrams can depict tasks as **active classes**; e.g., an **active class** is the root of a task thread.
- UML includes stereotypes of active classes **<<process>>** and **<<thread>>** to distinguish between processes and threads.

Thread Patterns

- **Dispatcher Pattern:** A single dispatcher thread receives requests from the OS and dispatches it onto worker threads.
- **Team (Replicated Worker) Pattern:** Worker threads pick up new requests independently.
- **Pipeline Pattern:** Each thread in the pipeline does part of the work; e.g., assembly line.

Thread Identification

- **Single event groups** - in simple systems, create a separate thread for each event.
- **Event source** - group events from a common source in the same thread.
- **Interface device** - encapsulate control of a specific device within a single thread.

Thread Identification (cont.)

- **Related vs. unrelated information** - group related information within a single thread; also called *functional cohesion*.
- **Timing characteristics -** group data items that arrive at the same rate.
- **Safety and reliability -** separate safety monitoring from actuation.
- **Purpose -** group items that have the same purpose.

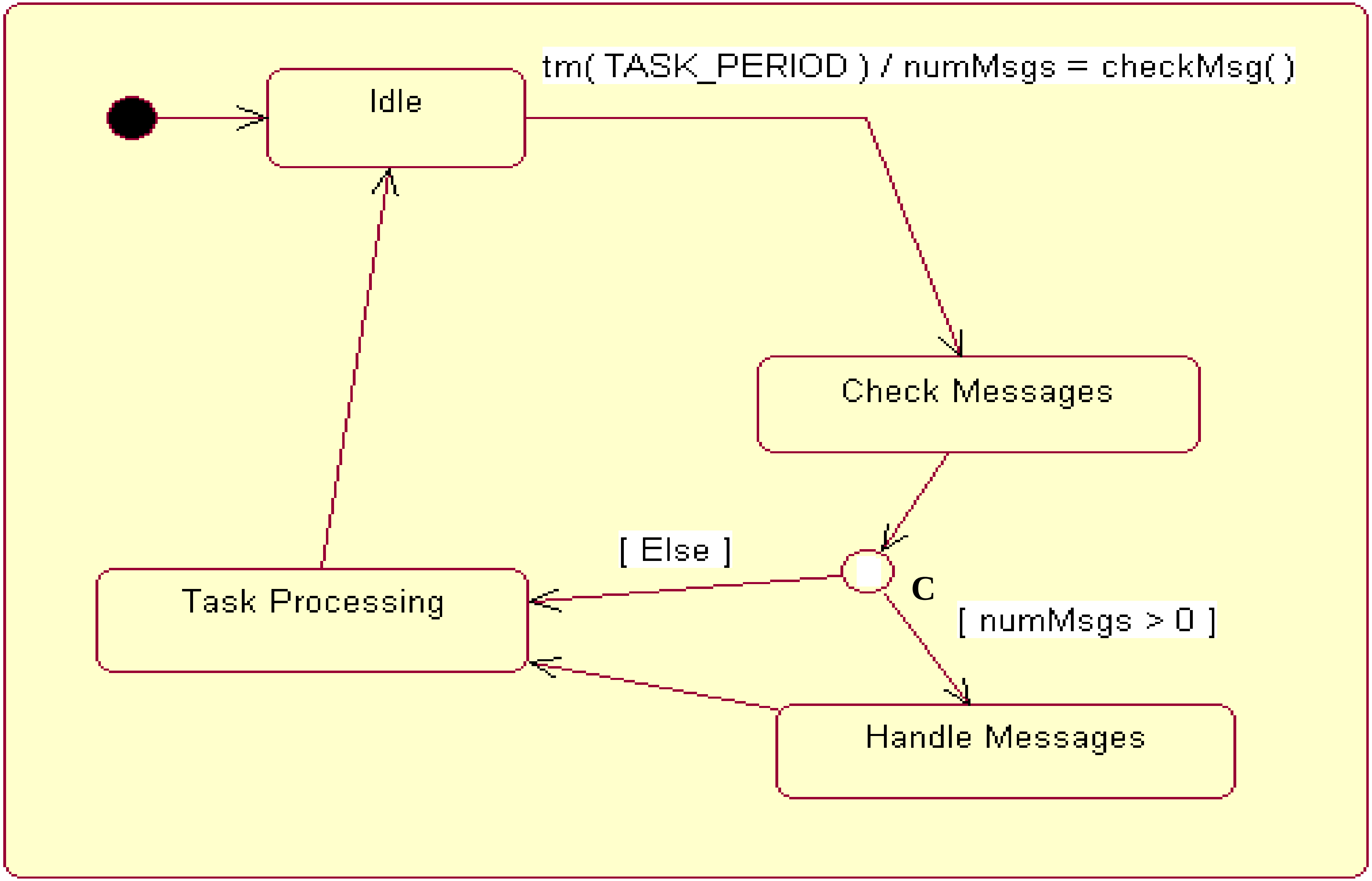
Assign Objects To Tasks

- After a set of tasks is identified, start populating the tasks with objects.
- Objects may appear in different tasks or as an interface between tasks.
- Most commonly, **active classes** are composites that create their component parts after creating the thread in which they will execute; the **<<active>>** stereotype is attached to the composite.

Task Processing

- **Event driven** - the active object runs an “event loop” and checks for messages in its message queue.
- **Time driven** - when the timer expires, the task awakens and performs a periodic function, or create an internal timeout (tm( )) event transition.

Periodic Active Object (Time Driven)



Task Communication

- Primary Reasons
  - **Share information** - data may need to be exchanged among tasks.
  - **Synchronize control** - the completion of a task may form a pre-condition for another task.
- Rendezvous - one method used for task communication.

Types of Rendezvous

If pre-conditions are not met, then:

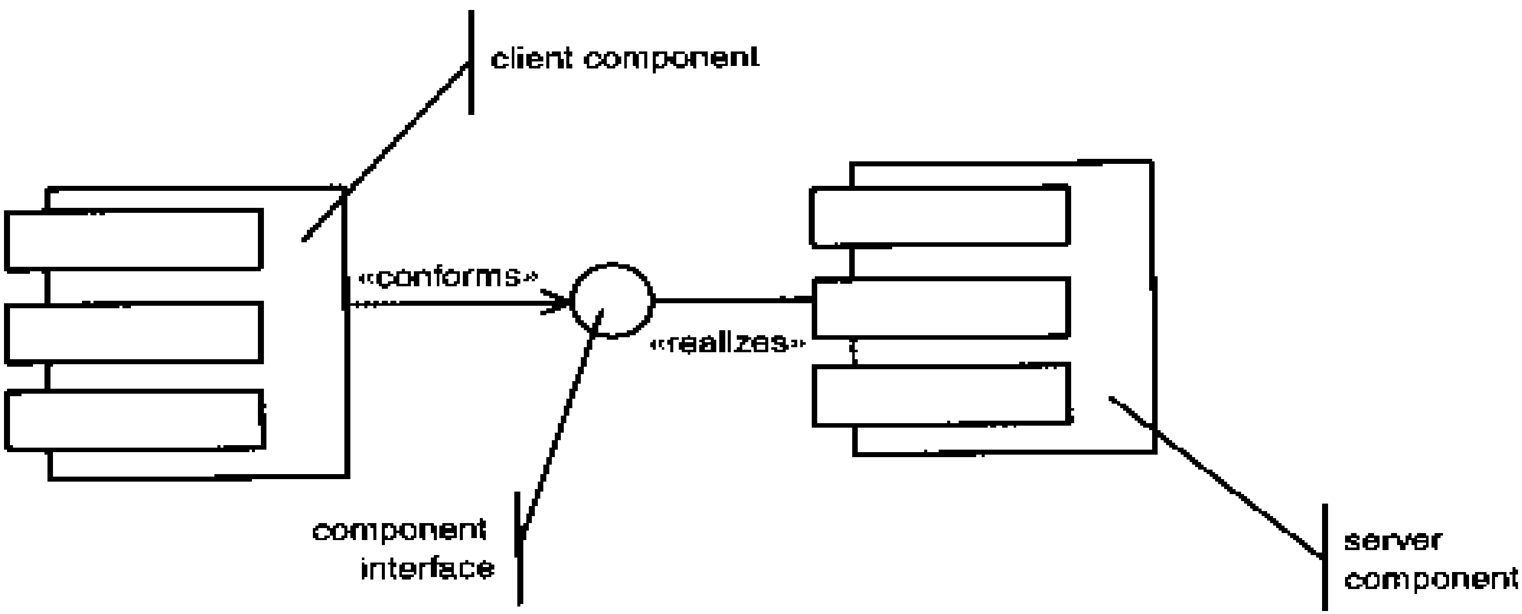
- wait indefinitely (**blocking rendezvous**),
- wait until task is ready or a specific time interval has elapsed (**timed rendezvous**),
- return immediately (**balking rendezvous**),
- or raise an exception and handle the failure as an error (**protected rendezvous**).

Synchronization

- Shared data access may need to be synchronized to ensure data integrity.
- Synchronization objects must handle:
  - preconditions
  - access control
  - data access
- A rendezvous object creates one lock object for each active object involved in a particular synchronization.

Component Model

- **Component diagrams** illustrate the organizations and dependencies among software components.



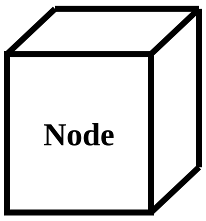
Component Classification

- Components can be classified as:
  - source code components,
  - run-time components, or
  - executable components.

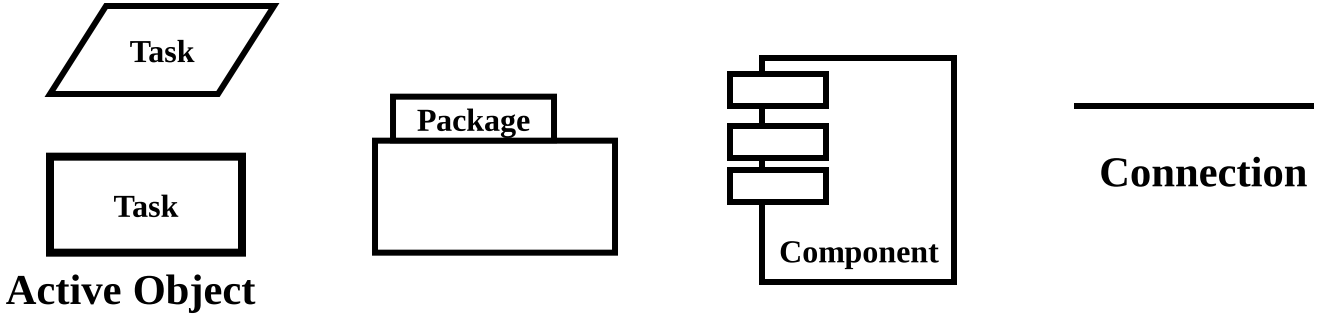
Deployment Model

- The **deployment diagram** shows the configuration of run-time elements and the processes (threads) living in them.
- The deployment diagram depicts all components distributed across the entire enterprise.

Deployment Diagram Notation



Nodes are of primary importance on a deployment diagram because they represent processors, sensors, actuators, displays, or any other physical object of importance to the software.



Safety/Reliability Model

- A **safe** system does not create accidents leading to injury, loss of life, or damage.
- A **reliable** system continues to function for long periods of time, even after some local failures.

Homogeneous Redundancy Pattern

- Uses identical channels (set of devices that handles cohesive set of data/control flows) to increase reliability.
- Uses voting policy, such as “majority wins” to detect and correct failures on minority channels.

Diverse Redundancy Pattern

The Diverse Redundancy Pattern can be implemented in several different ways:

- **Different but equal:** Redundant channels are implemented in totally different ways.
- **Lightweight redundancy:** A secondary channel ensures correctness of the primary by providing a "reasonableness" check.
- **Monitor-Actuator:** Monitors and actuators use different channels.

Watchdog Pattern

- A **watchdog** is a component that receives messages from other components on a periodic or sequence-keyed basis.
- If messages are received too late or out of sequence, then the watchdog initiates a recovery action.

Rational Rose RealTime (Rose RT) Views

- **Views:** each view contains a number of diagrams describing a certain aspect of the system.
- There are four types of views:
  - Use-Case View
  - Logical View
  - Component View
  - Deployment View
- **Diagrams:** are graphs describing the contents in a view.

UML Views

- **Use-Case View:**
  - describes system functionality as perceived by *external actors* which interact with the system (users or other systems)
- **Logical View:**
  - describes how system functionality is provided within the system - mostly used by developers
  - describe the **static structure** and **dynamic behavior** and other properties such as persistence and concurrency.
  - static structure: described by *class* (*object) diagrams*
  - dynamic behavior: described by *state, sequence, collaboration* and *activity diagrams.*

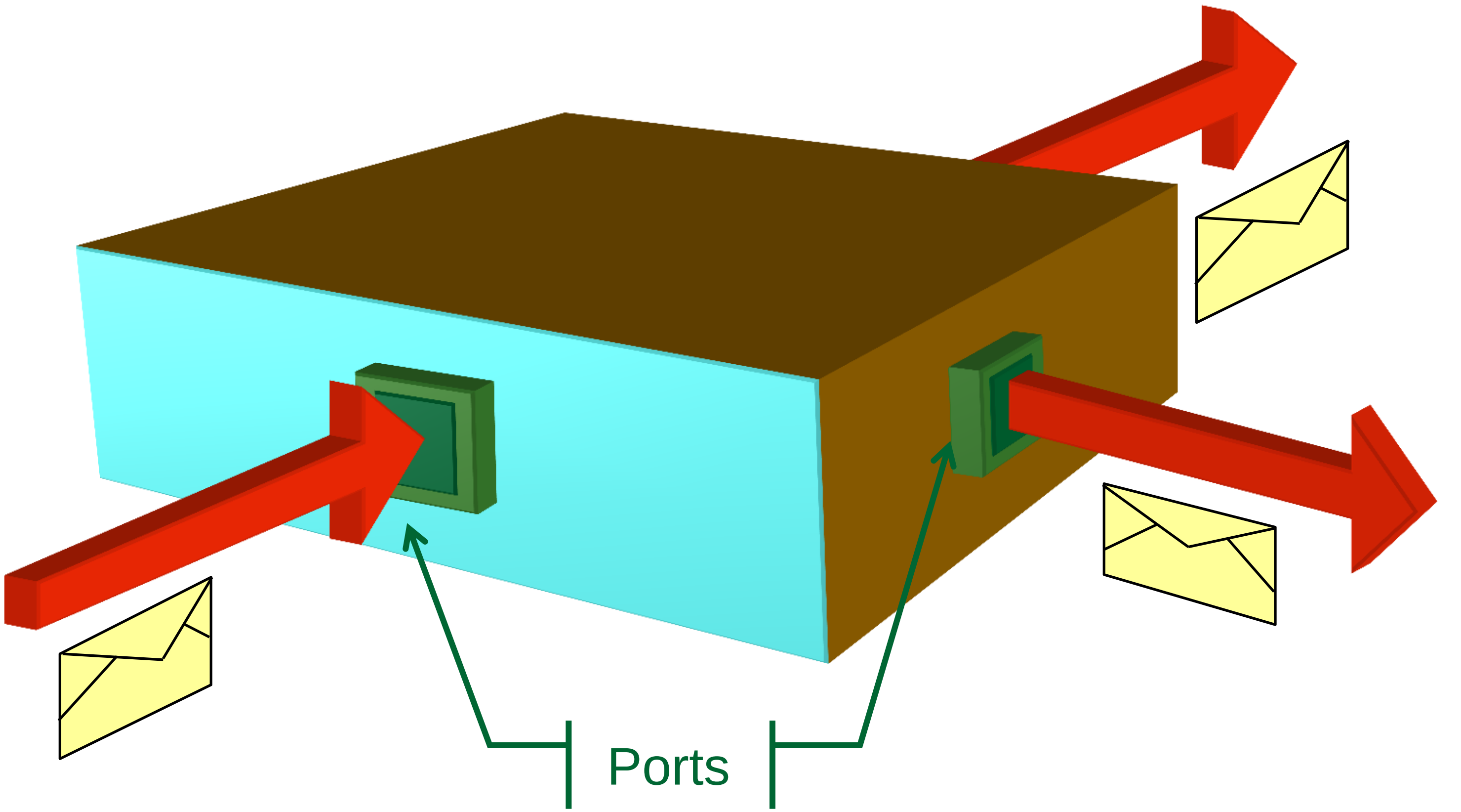
UML Views

- **Component View:**
  - description of the implementation modules and their dependencies (used mainly by developers)
  - contains *component, package diagrams*
- **Deployment View:**
  - shows the physical deployment of the system (processors, devices) and their interconnections
  - contains *deployment diagrams.*

Real-Time UML Constructs

- For Modeling **Structure**
  - capsules (capsule classes)
  - ports
  - connectors
- For Modeling **Behavior**
  - protocols
  - state machines
  - time service

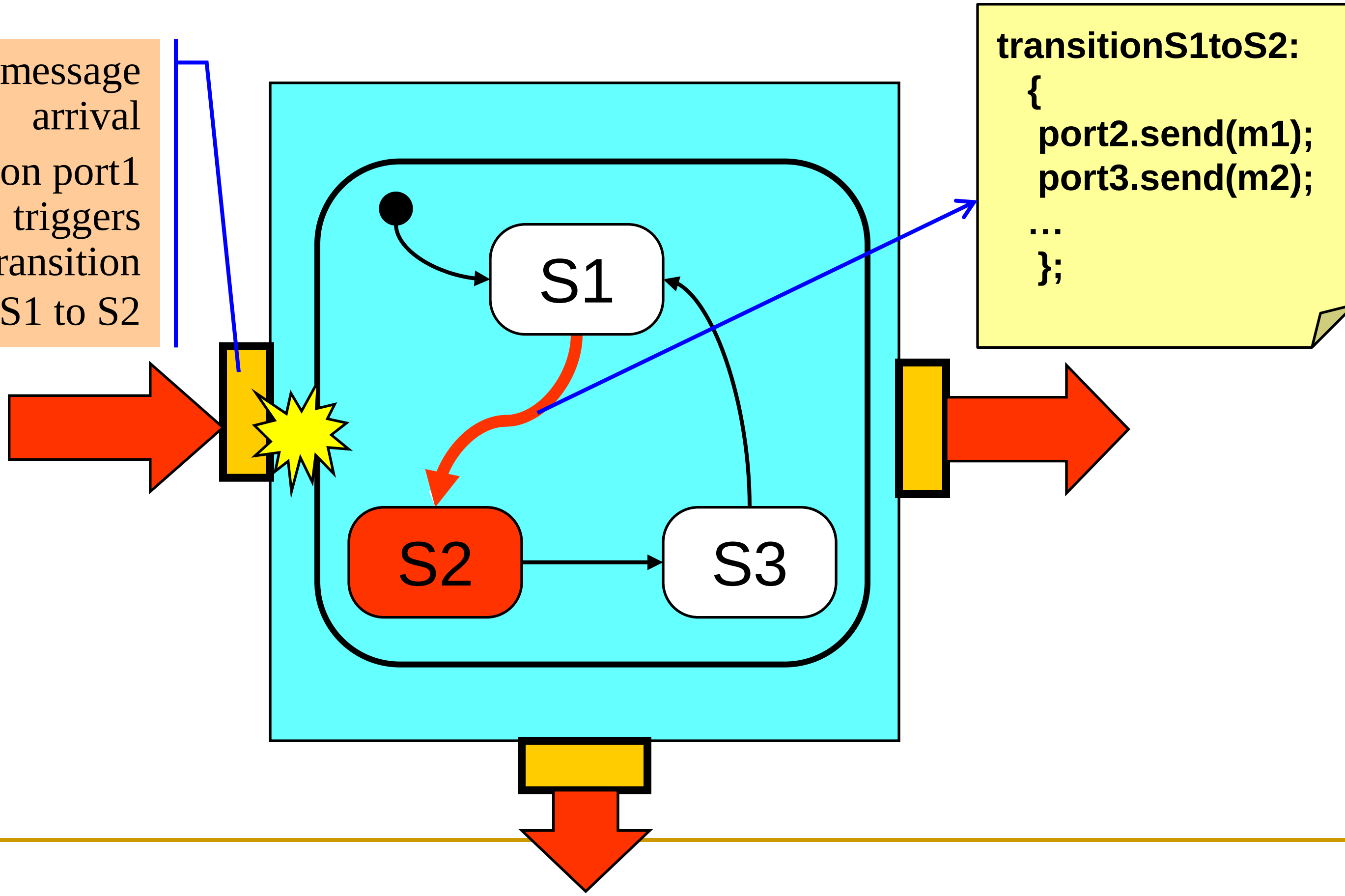
Capsules: Active Objects



Capsules: Behavior

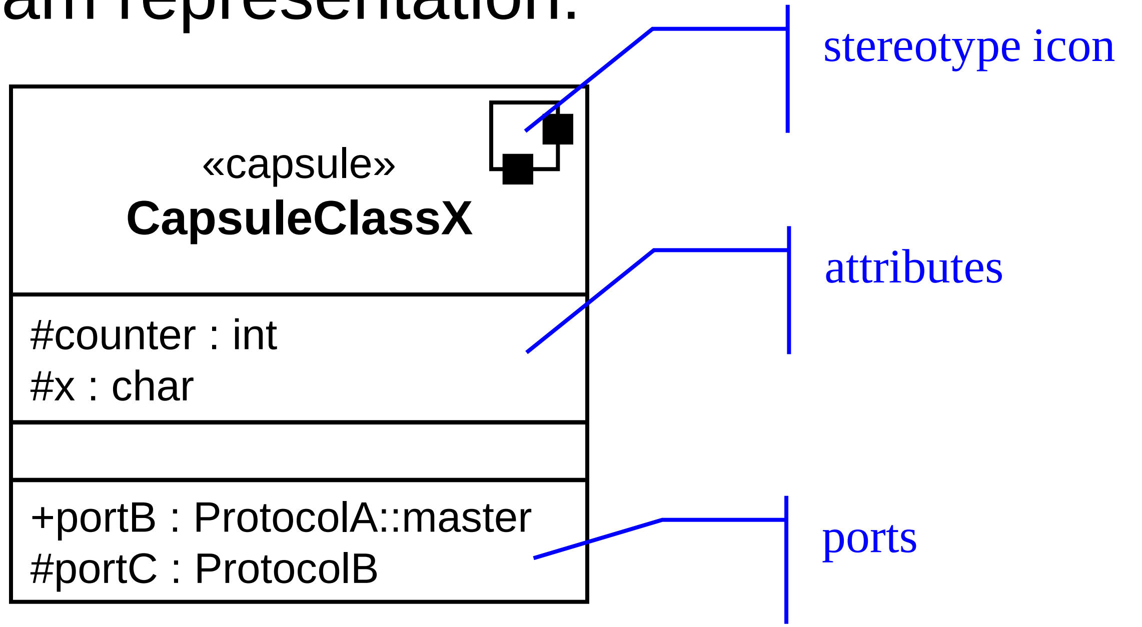
- Optional hierarchical state machine

message arrival on port1 triggers transition S1 to S2



Capsules: UML Modeling

- Stereotype of Class concept **«capsule»** with specialized (executable) semantics
  - represent independent flow of control (active classes)
  - used to represent the architecture of a system
- Class diagram representation:



Classes vs. Capsules

- Communication:
  - Classes: Public operations
  - Capsules receive messages through public **ports** which understand protocols.
- Attributes:
  - Classes: Public, private, and protected.
  - Capsules only have private attributes to enforce encapsulation.

Classes vs. Capsules

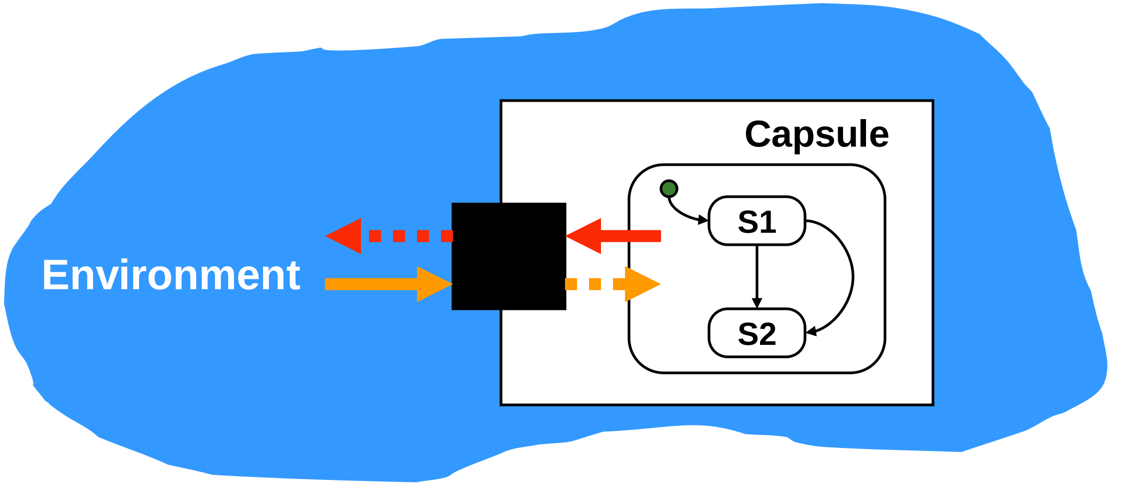
- Behavior:
  - Classes: Method implementation.
  - Capsules: Defined by state machines which run in response to the arrival of signals.

Ports

- Boundary objects for a capsule instance.
- Unlike an interface (which is a behavioral thing), a port includes both structure and behavior.
- Each port plays a specific role in a protocol.
- The protocol defines the valid flow of information (signals) between connected ports of capsules.

Ports: Boundary Objects

- Fully isolate a capsule's implementation from its environment (in both directions)



Ports are created and destroyed along with their capsule

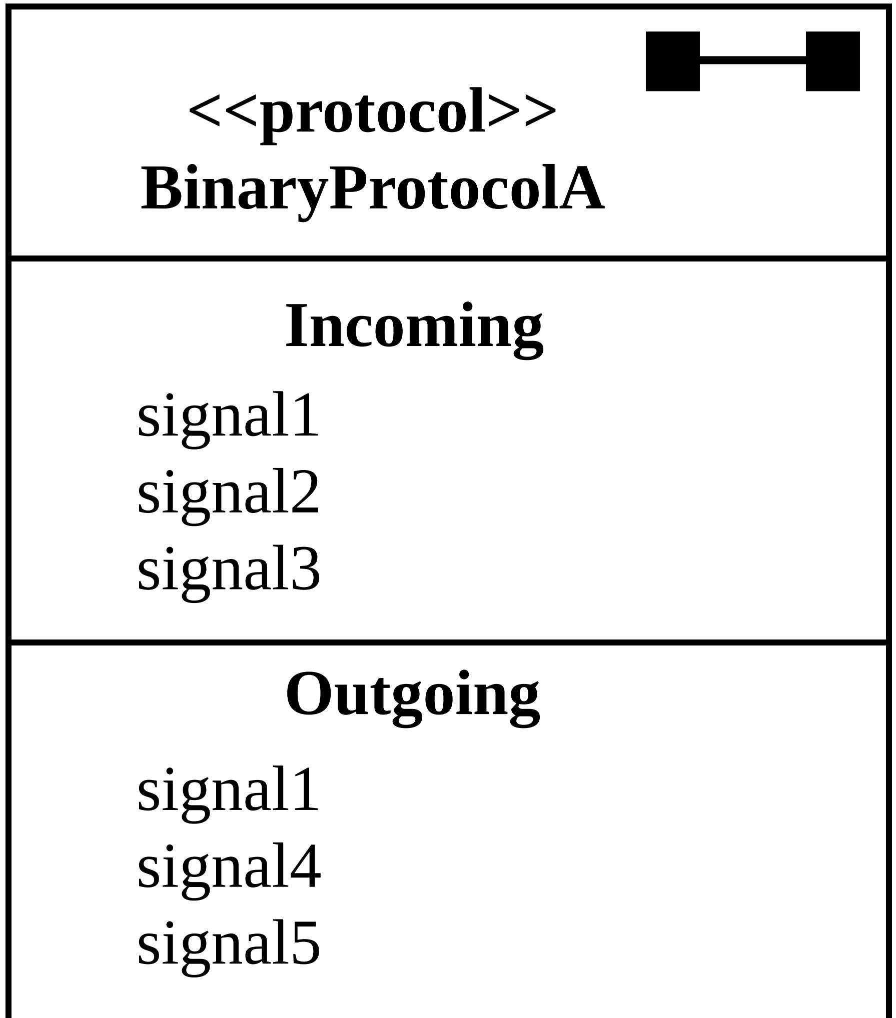
Types of Ports

- Viewed from the outside, ports present the same object interface, and they cannot be distinguished except by their identity and protocol role.
- Viewed from inside the capsule, they can be one of two kinds:
  - **relay ports** - connected to sub-capsules
  - **end ports** – directly connected to the capsule's state machine

Protocol

- A **protocol** is a specification of desired behavior to take place over a connector.
  - It is pure behavior and does not specify any structural properties.
- **Binary protocols**, involving only two participants, are the most common.
  - For binary protocols, only one role, called the **base role**, needs to be specified.
  - The **conjugate role** can be derived from the base role.

Protocol


<<protocol>>
BinaryProtocolA
Incoming
signal1
signal2
signal3
Outgoing
signal1
signal4
signal5

Protocol Roles

- Specifies one party (the base party) in a protocol

Incoming signals

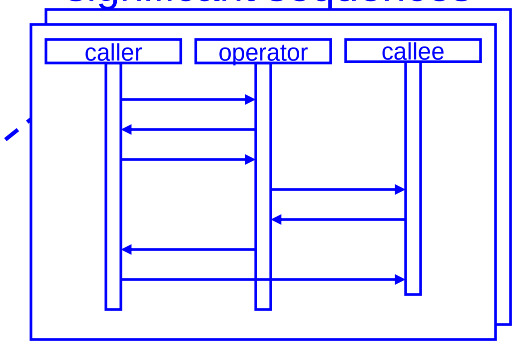
signal	*source*
call	**caller**
number	**caller**
ack	**callee**

OperatorRole

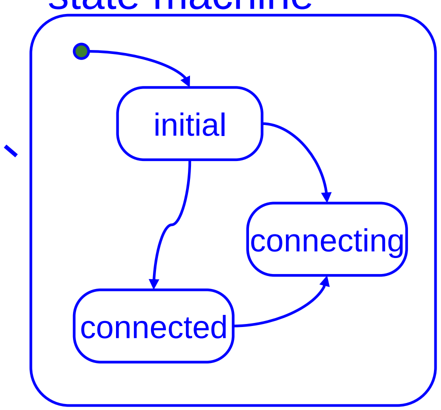
Outgoing signals

signal	*target*
call	**callee**
transfer	**caller**
ack	**caller**

significant sequences



state machine



Protocol Refinement

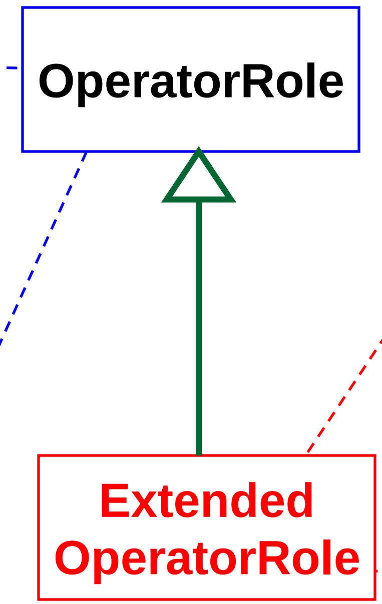
- Using inheritance

Incoming signals

signal	*source*
call	caller
number	caller
ack	callee

Outgoing signals

signal	*target*
call	callee
transfer	caller
ack	caller



Incoming signals

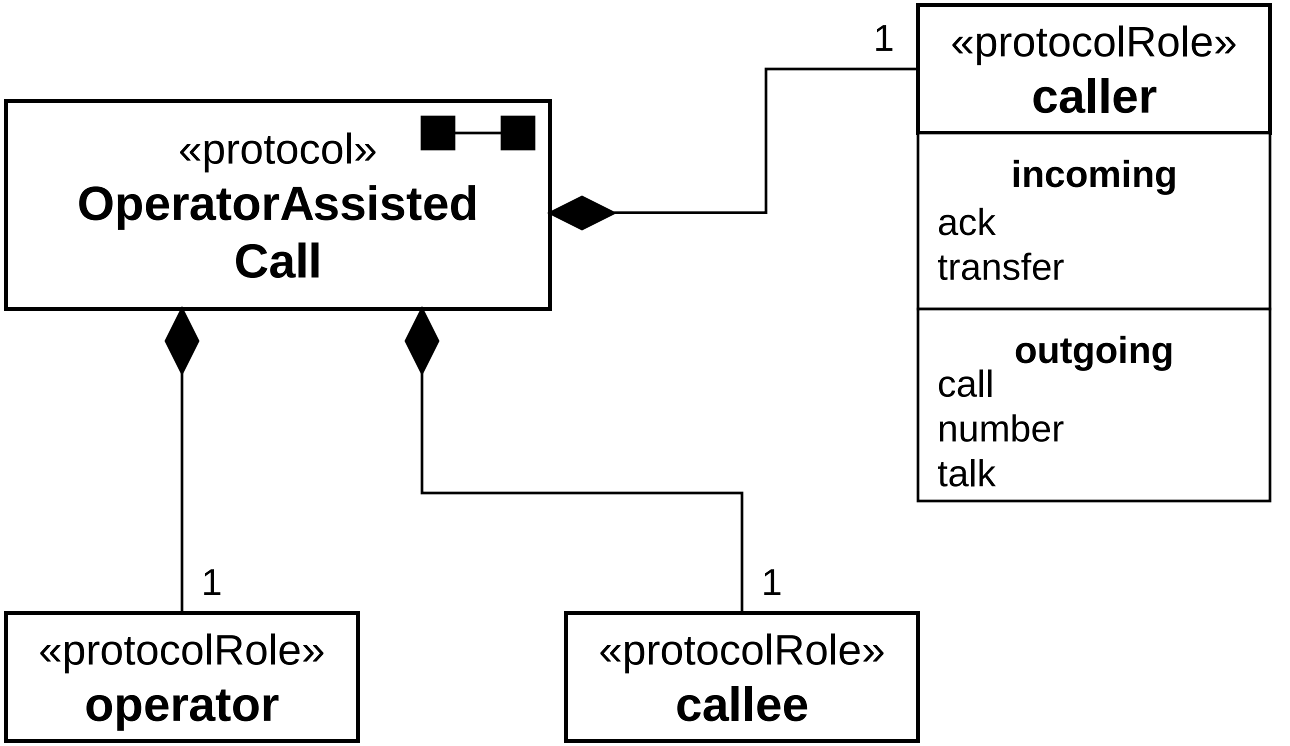
signal	*source*
call	caller
number	caller
ack	callee
reply	*caller*

Outgoing signals

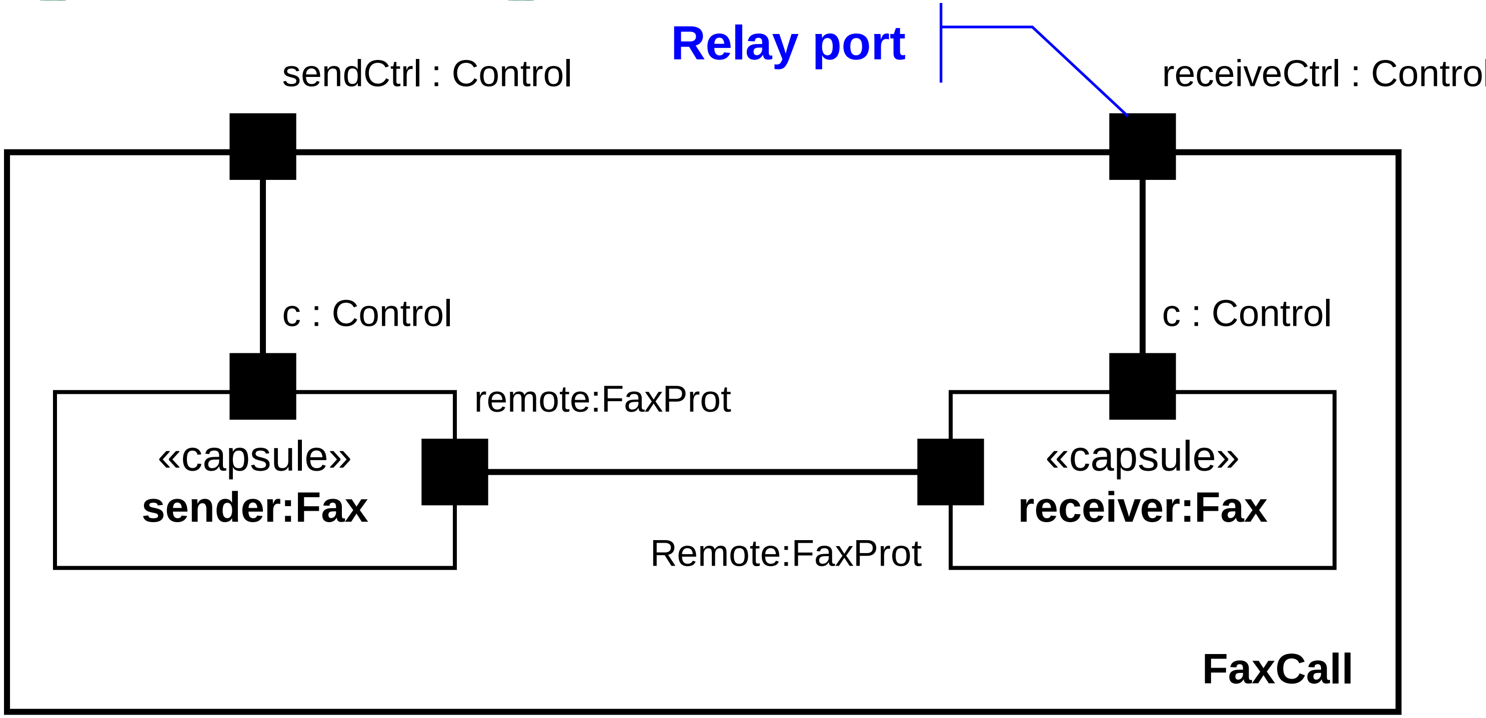
signal	*target*
call	callee
transfer	caller
ack	caller
query	*caller*

Protocols: UML Modeling

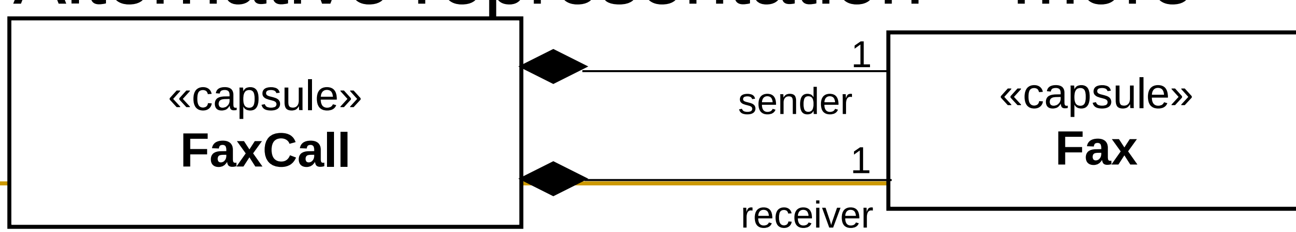
- Collaboration stereotype: **«protocol»**
- Classifier Role stereotype: **«protocolRole»**



Capsule Composition



- Alternative representation – more



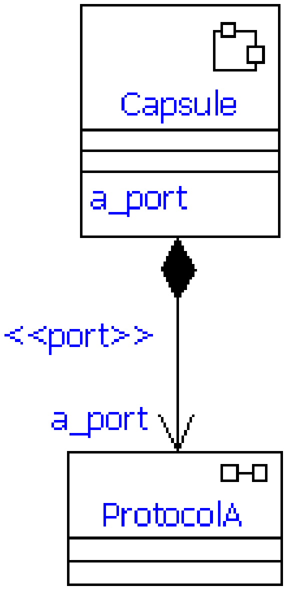
Classification of Ports

- **Visibility**
  - **Public ports:** part of a capsules interface - located on the boundary of a capsule structure
  - **Protected ports:** not visible from the outside.
- **Connector type**
  - **Wired ports:** connected statically by a *connector* to other ports
  - **Non-Wired ports:** connected dynamically - used to model dynamic communication channels.

Classification of Ports

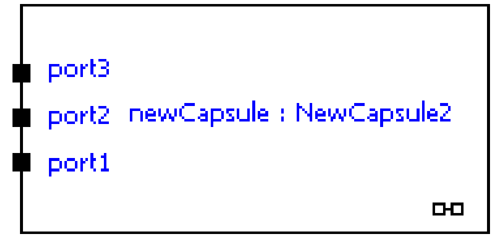
- **Message processing**
  - **End ports** are the ultimate destination of all messages sent by other capsules, which are processed by the state machine of the capsule owning the end port. End ports can be *public* or *protected*, *wired* or *non-wired*.
  - **Relay ports** are implicitly *public* and *wired*, and are used for connections that funnel messages directly to protected capsule components without being processed by the owner of the relay port. If a relay port is not connected to an internal component, all messages arriving to the port are lost.

UML-RT Notation for Ports



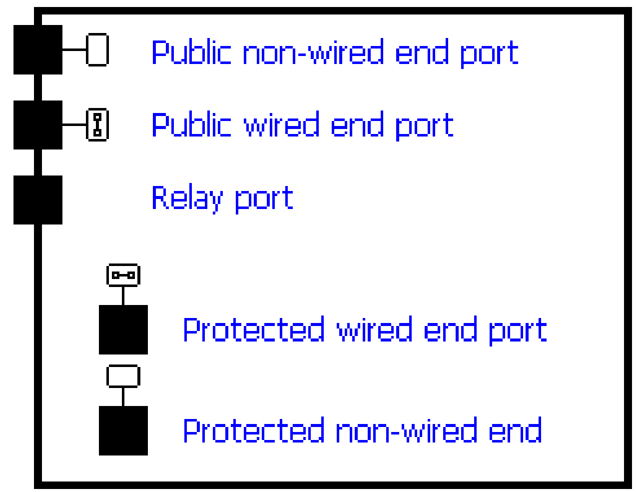
a) Class diagram

Note. All the ports shown here implement the base role of their protocol.



b) Capsule role view

(only public ports are shown)



c) Capsule structure diagram

(all ports are shown)

System Ports

- There are four kinds of **system ports** that are used for accessing features of the run-time system:
  - **Frame**: used for dynamic creation, destruction, import and export of capsules at run-time
  - **Timing**: used to access the timing service (setting different timers)
  - **Log**: used to access the log service (printing logging messages)
  - **Exception**: used to access the exception service (ability to define custom policies to recover from exceptions)
- How to access different run-time system features through system ports:
  - create a non-wired port of the system port type required
  - invoke required operation on the port:

portName.functionName (args)

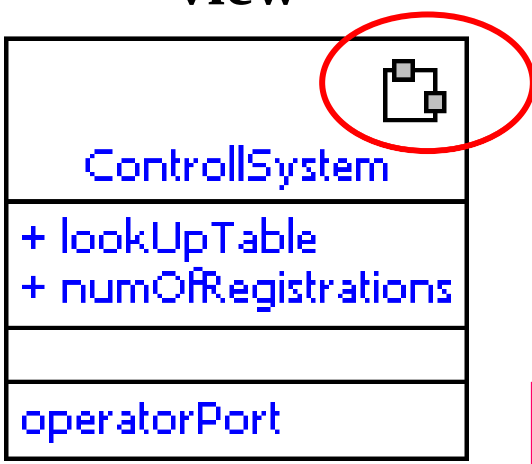
Summary on Capsules

- **Capsules** are the fundamental modeling element of real-time systems. A capsule represents an independent flow of control in a system.
- Similarities of capsules to classes:
  - Capsule can have *attributes.*
  - Capsules may also participate in *dependency*, *generalization*, and *association* relationships.
- Differences between capsules and classes:
  - **Capsule structure:** represented as a network of collaborating capsules (a *specialized UML collaboration diagram).*
  - **Capsule behavior:** triggered by the receipt of a signal event, not by the invocation of an operation.

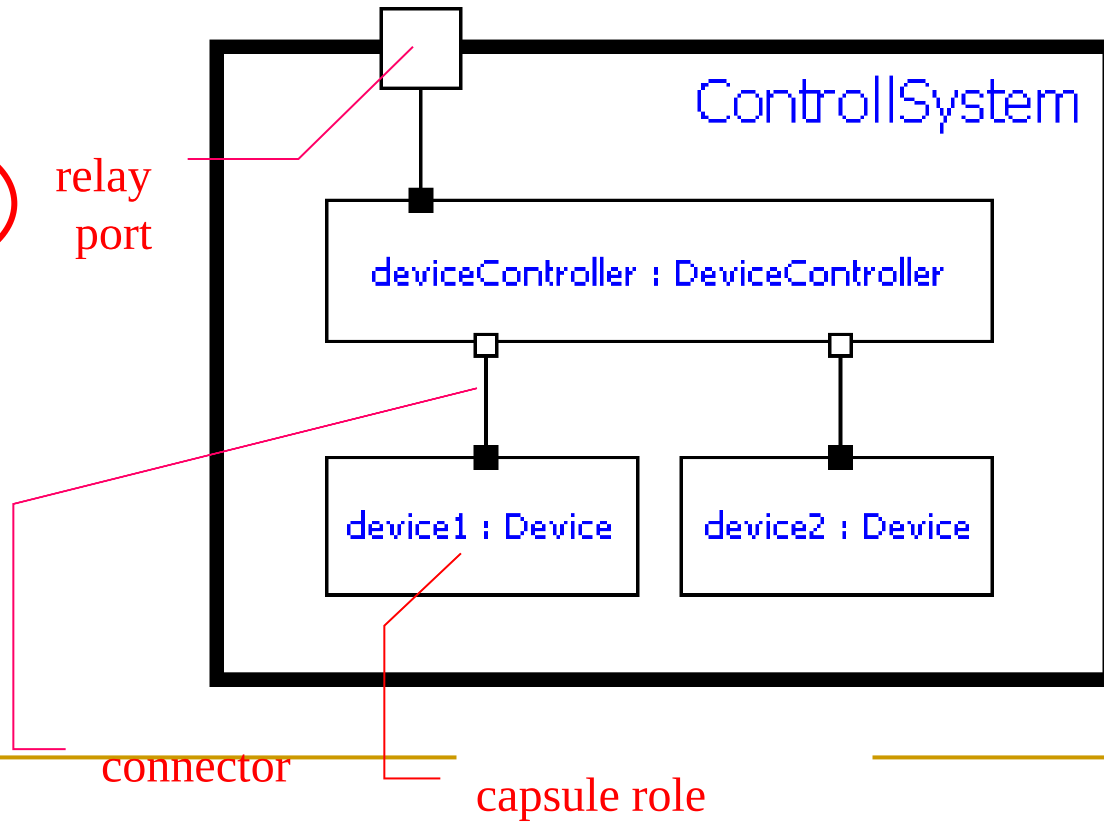
UML-RT Notation for Capsules

Since a capsule is a stereotype of a class, the stereotype icon appears in the name compartment of the class rectangle.

Class diagram view

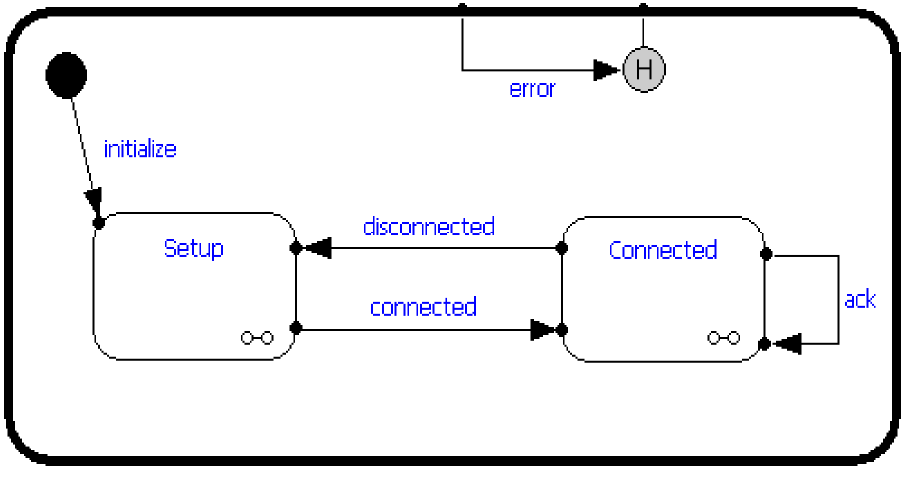


Capsule structure diagram view



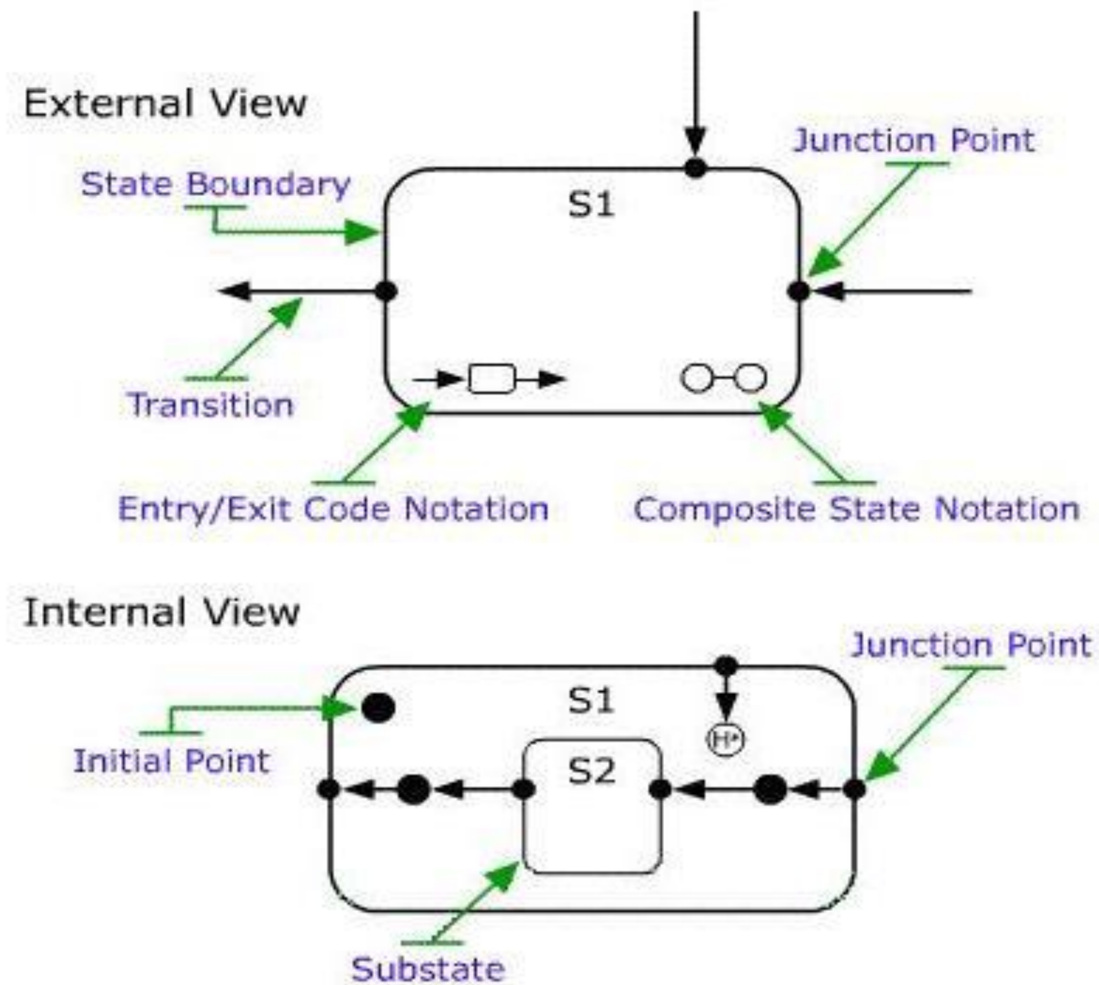
State Diagrams

- A *state machine* is a graph of *states* and *transitions* that describes the response of an object of a given class to the receipt of outside stimuli.
- The states are represented by state symbols and the transitions are represented by arrows connecting the state symbols.
- States may also contain subdiagrams, or other state machines which represent different hierarchical state levels.



States: external and internal view

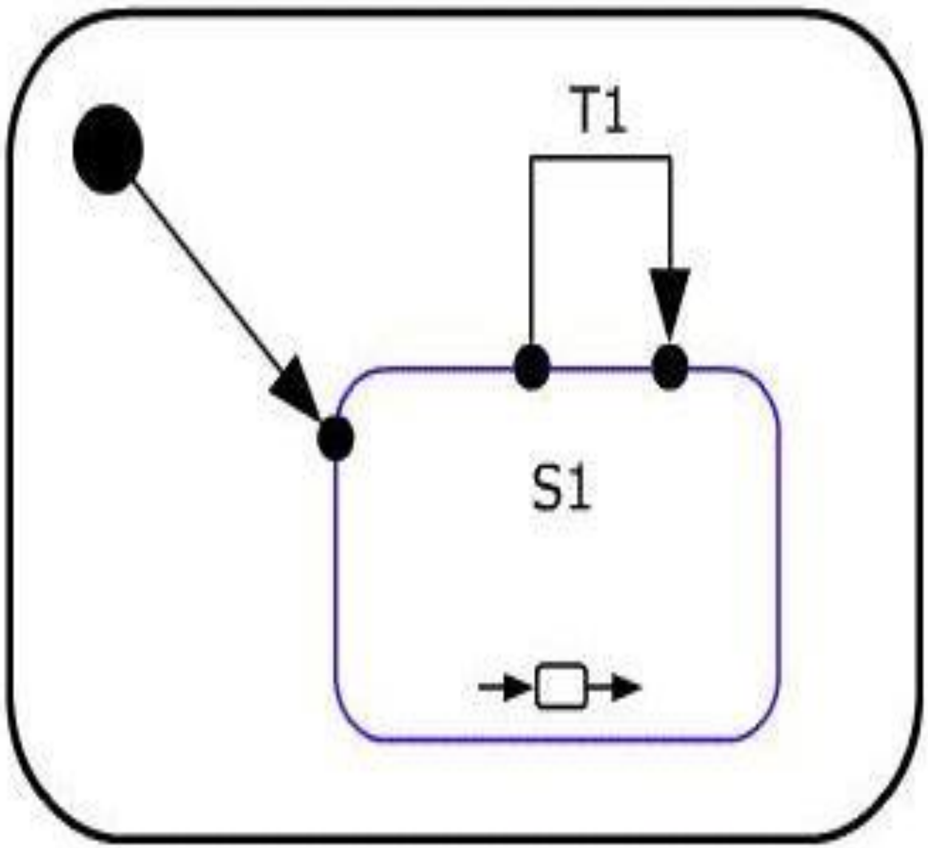
A state is a period during the lifetime of an object where it is ready to receive events.



State actions: entry and exit actions

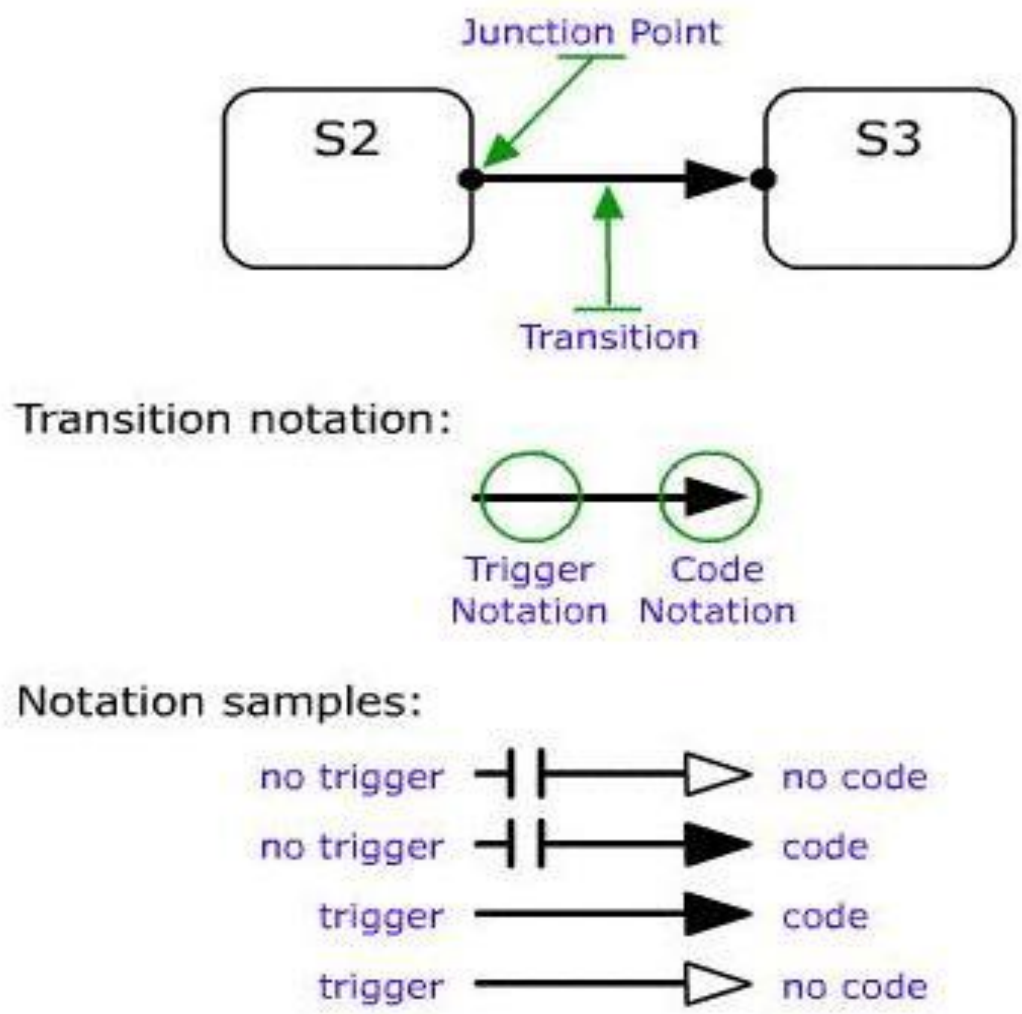
- assume S1 has entry and exit action
- when T1 fires:
  1. exit action S1
  2. T1 taken
  3. entry action S1
- entry and exit action only run if a transition fires

Example: Entry and Exit Code

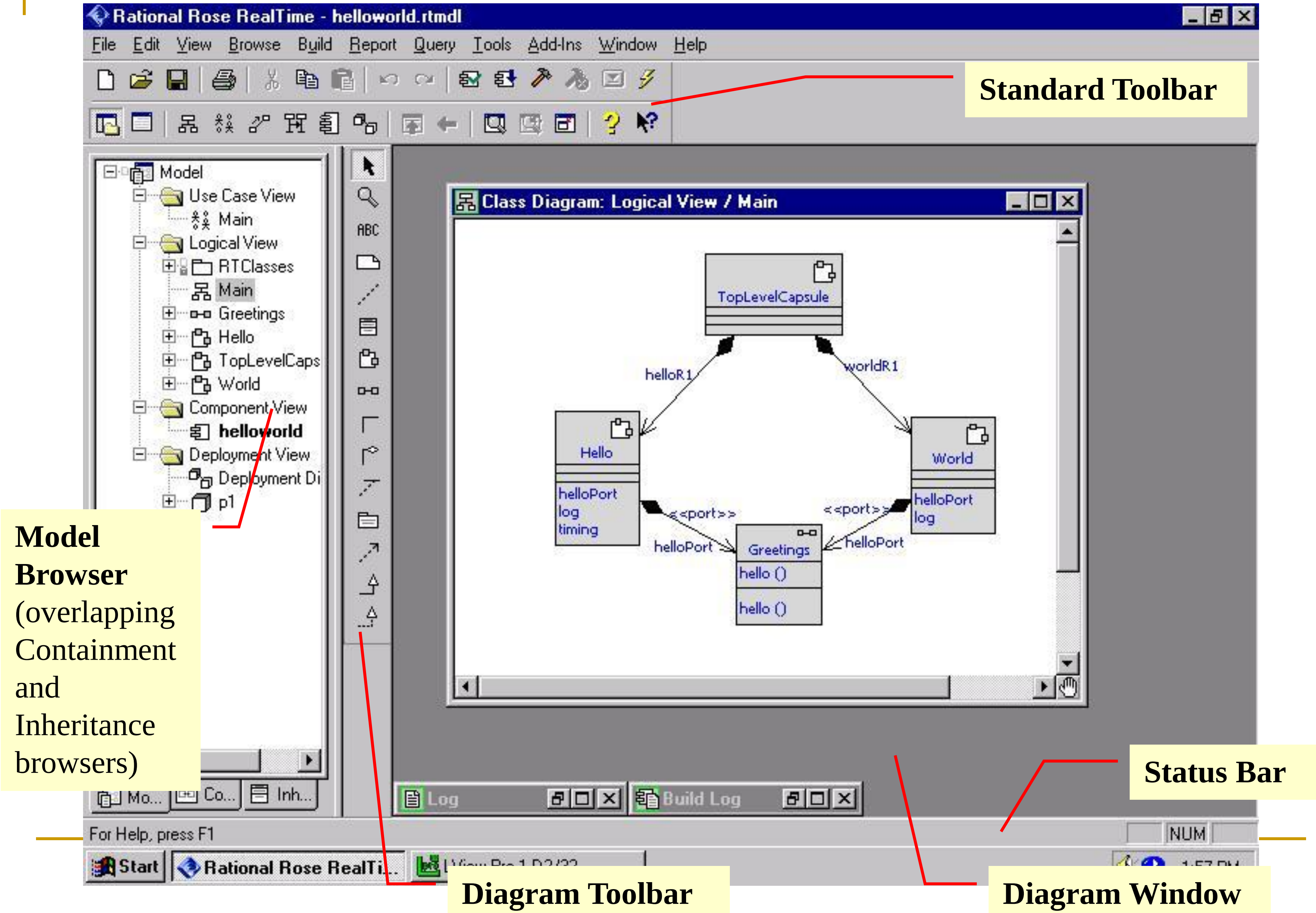


Transitions

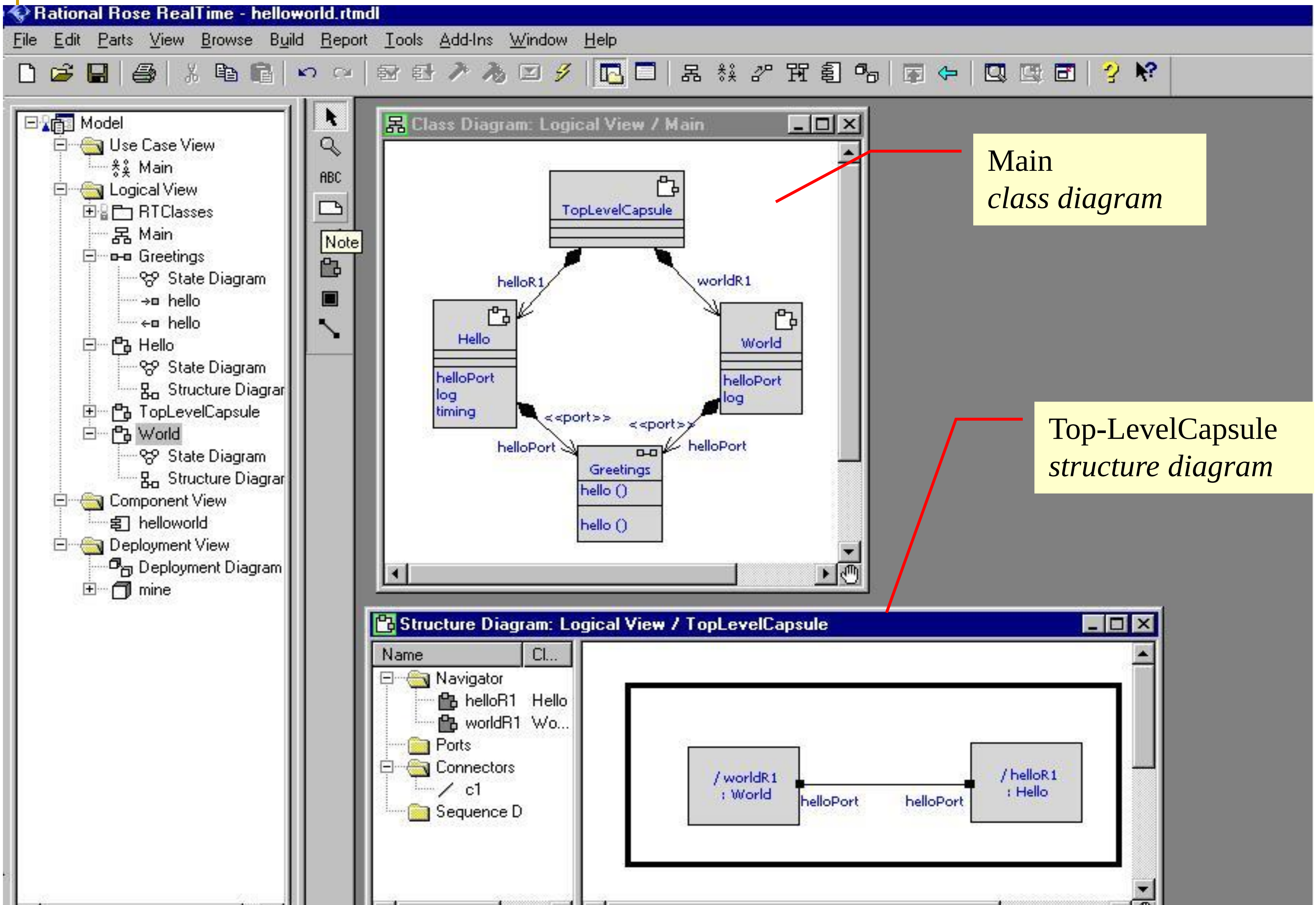
A transition specifies that when an object in a source state receives a specified event, and certain conditions are satisfied, the behavior will move from the source state to the destination state.



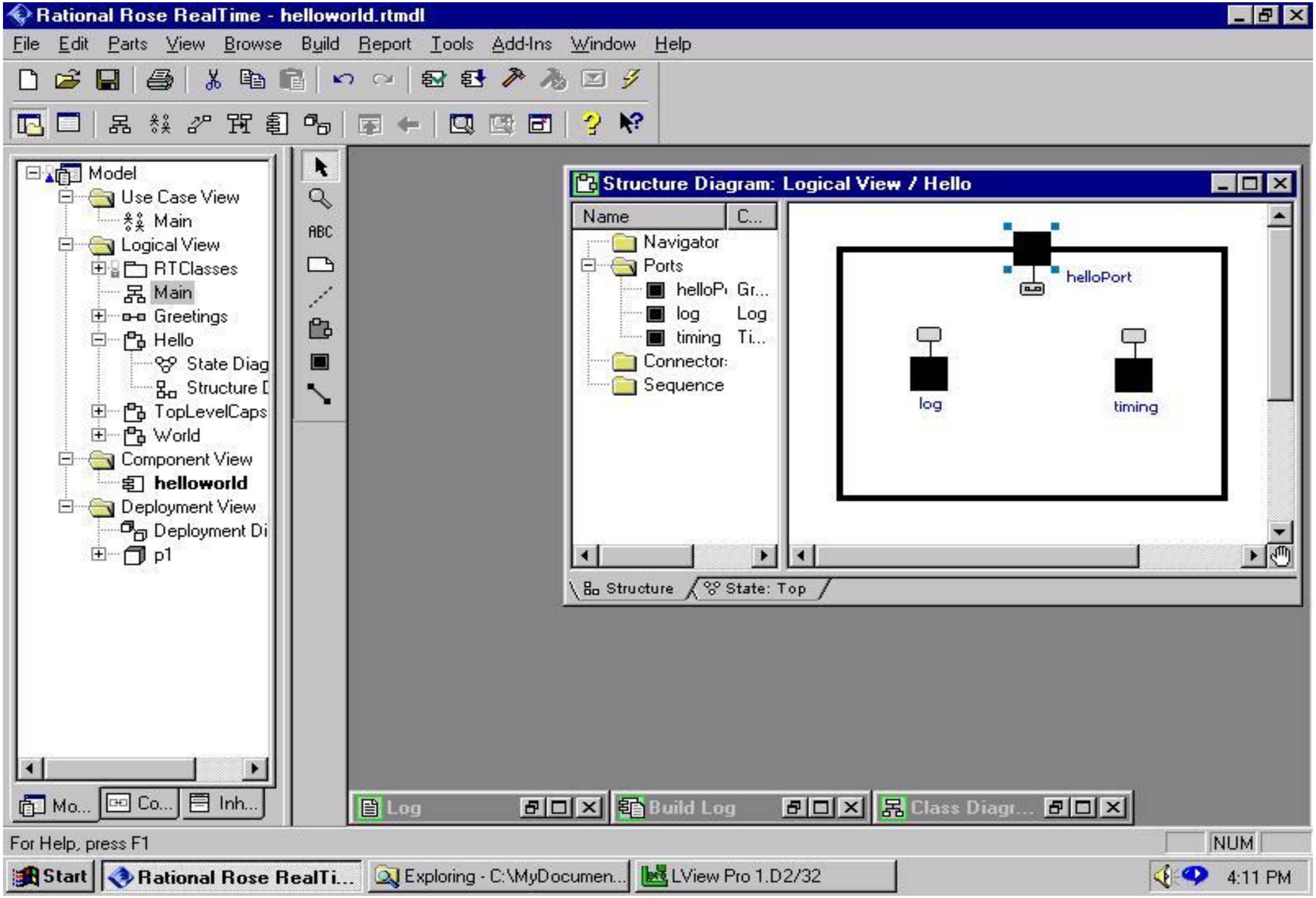
Rational Rose RealTime



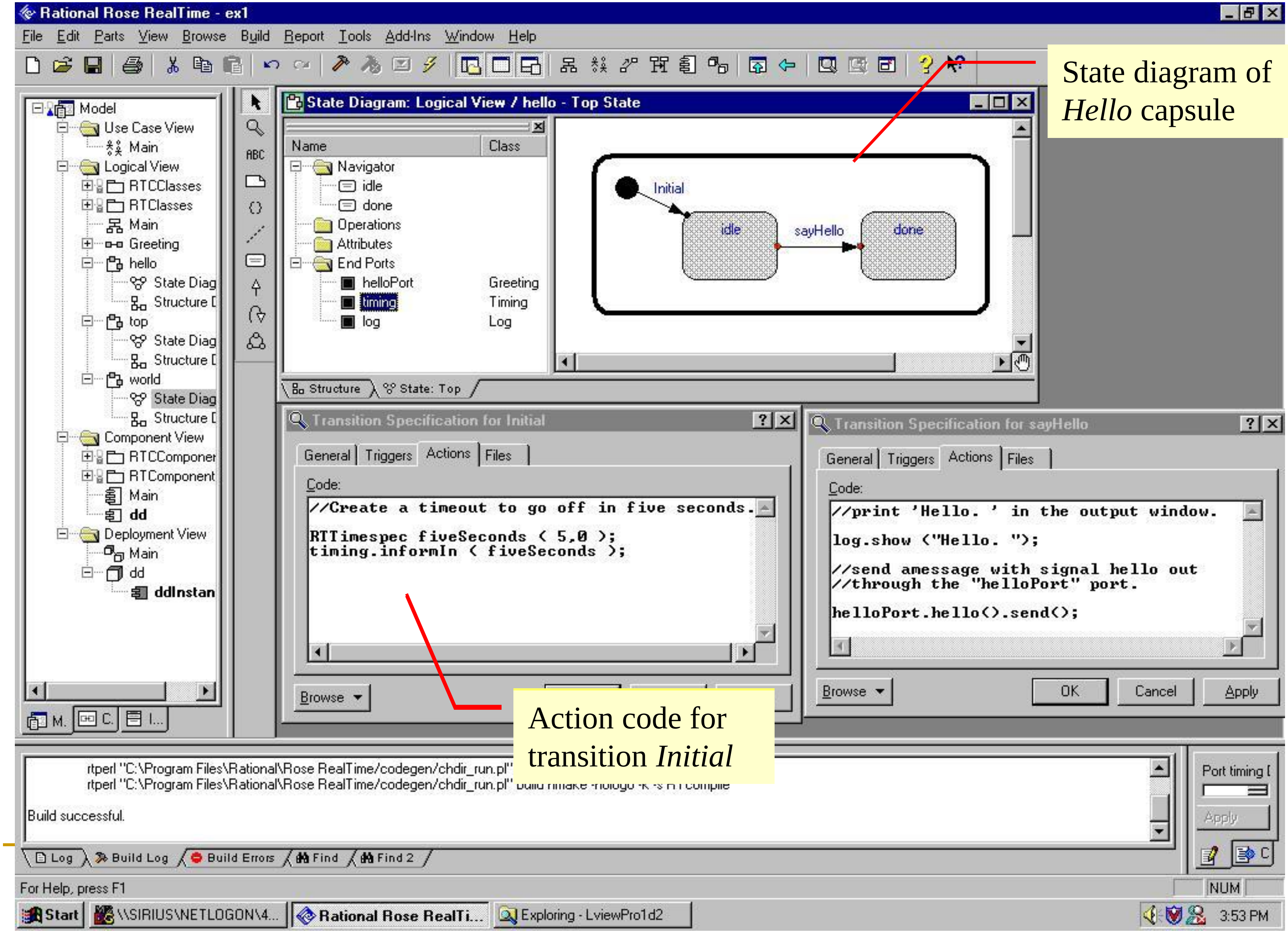
Class Diagram and Structure Diagram



Structure Diagram for Hello Capsule



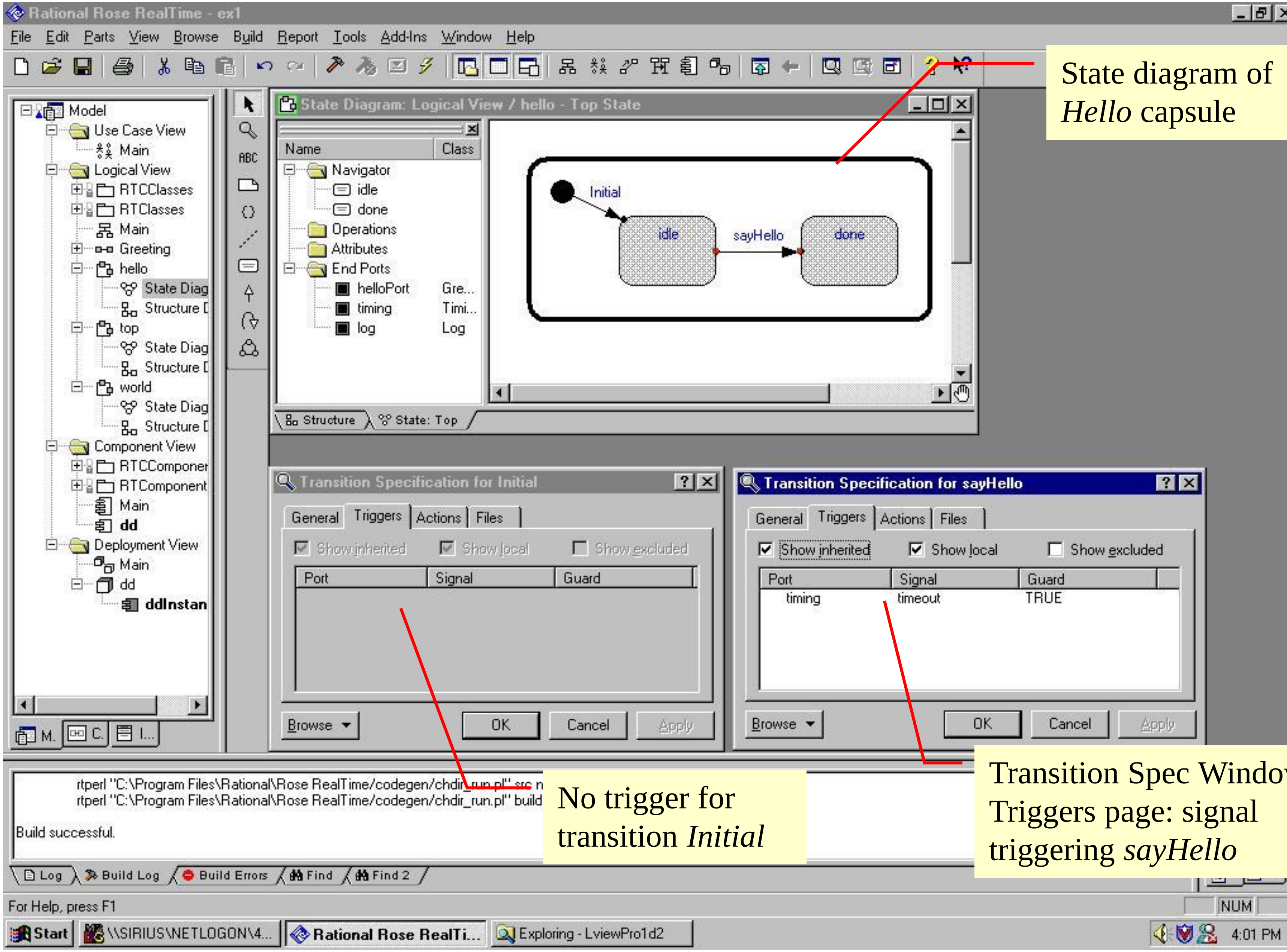
Hello State Diagram: transition actions



State diagram of *Hello* capsule

Action code for transition *Initial*

Hello State Diagram: transition triggers

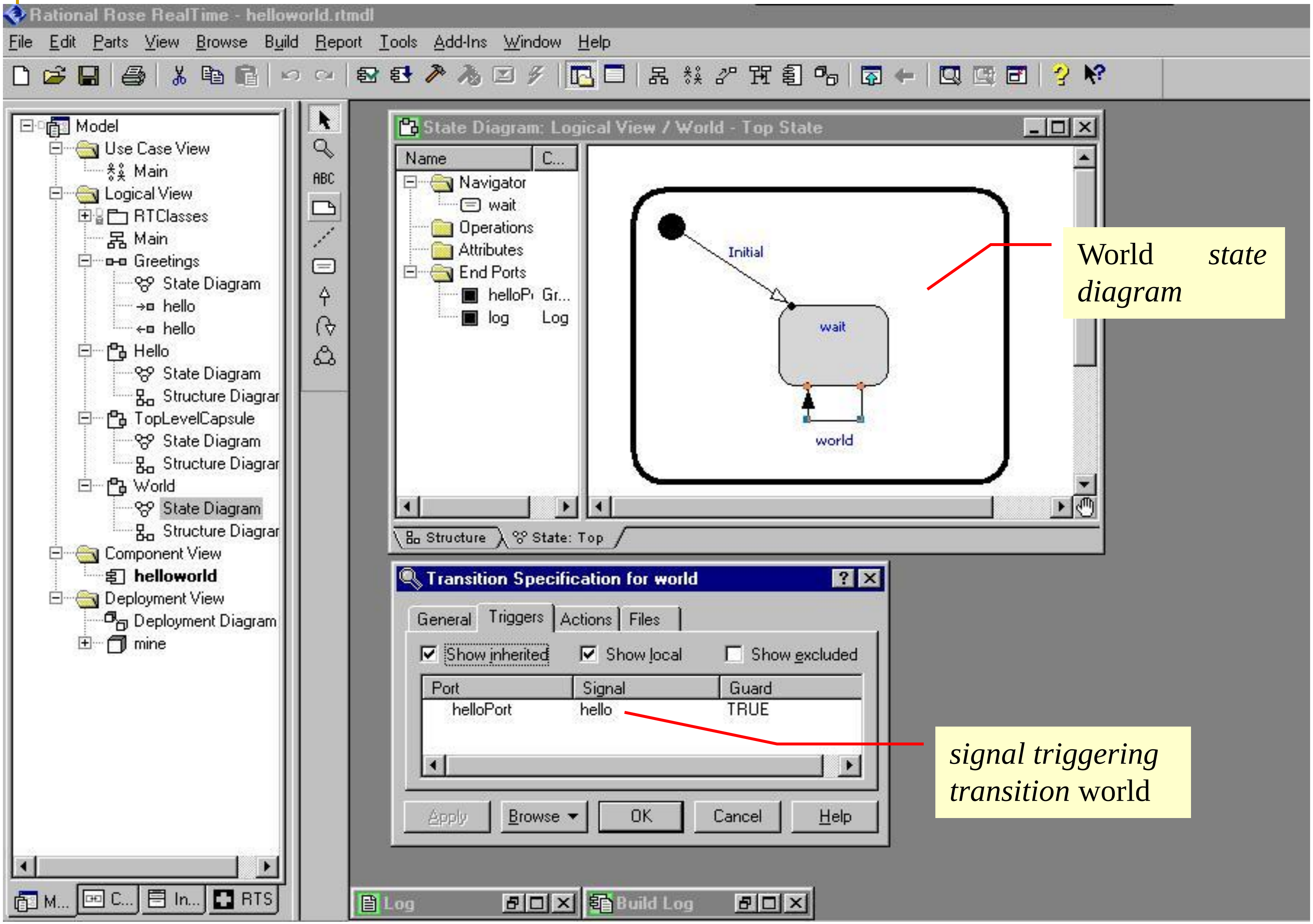


State diagram of *Hello* capsule

No trigger for transition *Initial*

Transition Spec Window, Triggers page: signal triggering *sayHello*

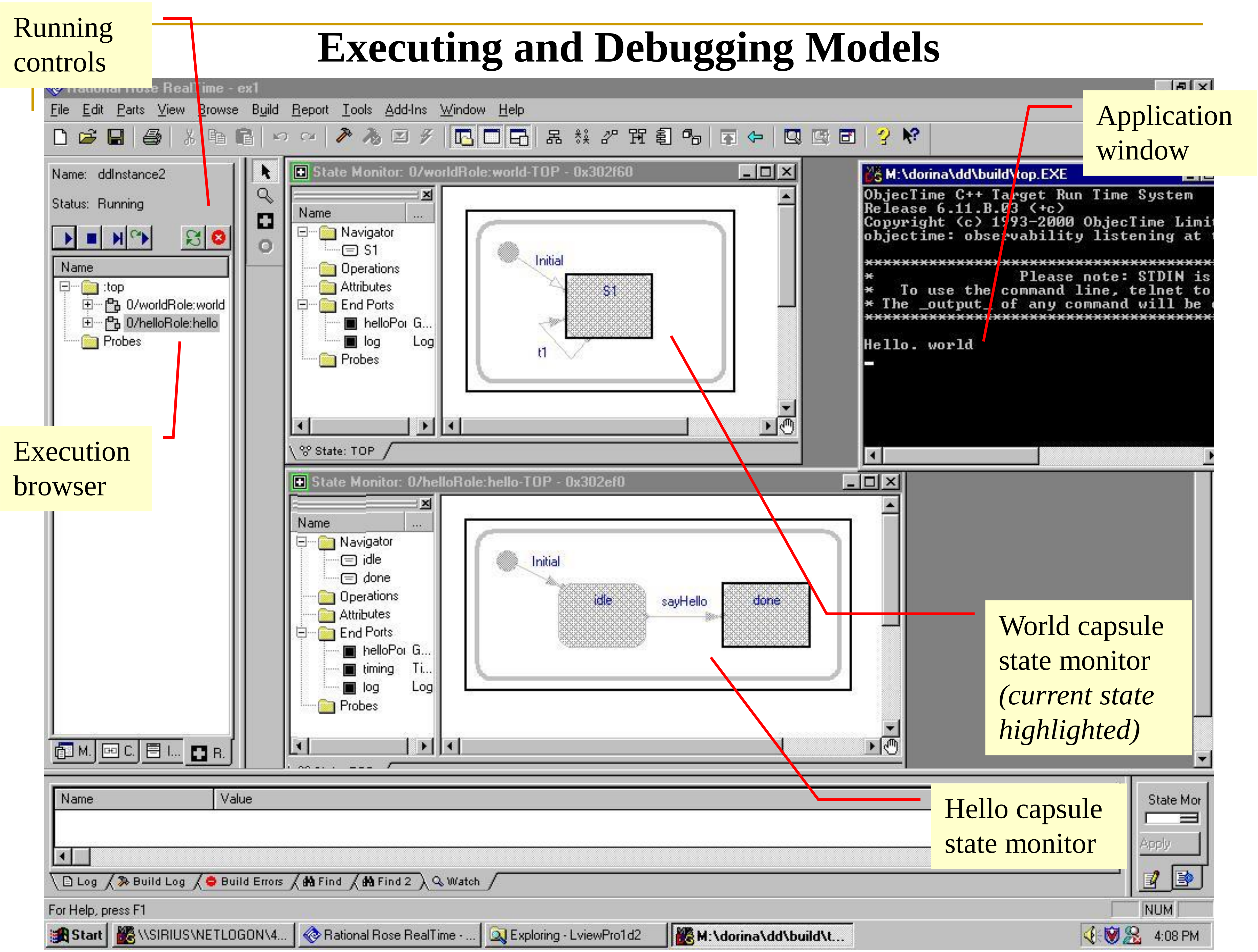
World State Diagram: transition triggers



World *state diagram*

signal triggering transition world

Executing and Debugging Models



Summary

- Next time: TimesTool
- Then: AADL and OSATE