Modeling and Programming of Distributed Real-Time Systems

TEMPUS S_JEP 8333-96

Rational

- Greater care during analysis to guarantee real-time constraints
- Special operating systems and development tools required
- Machine dependencies cause maintenance difficulties
- Difficulty of accurate worst-case prediction
- Resource reservation anomalies
- Problems of scheduling policy

Contents

- Introduction
- Representation
- Modeling and Programming
  - Real-Time ANSA
  - ROOM - Real-Time O-O Modeling
  - Real-Time JAVA

Introduction

- Large-Scale Systems
- Real-Time Distributed Systems
- System Behavior
- Problems
  - Predictability
  - User Control
  - Timeliness
  - Mission Orientation
  - Performance

Representation

- Real-Time Structured Analysis
  - Structured analysis (SA) extended by control flow diagrams
- Petri Nets
  - Using formal Petri nets for representing system behavior

- Hierarchical Graphic System Design Capture and Simulation Method
  - Hierarchically structured combination of RT-SA and Petri net methods
- System Requirements Engineering Methodology
  - Representation of behavior using time scenarios
- Z Formal Specification Language
  - Mathematical language for specifying systems independently of implementation

- Communicating Sequential Processes
  - Representing system as communicating processes of actions
- Modechart
  - Specifying system's timing properties by modes, conditions, events and actions
- Viewchart
  - Hierarchically composed statecharts with own namespace called views

- Design Capture Views
  - Partitioning system design into views by defined key aspects
- Use Case Maps
  - Making scaleable scenarios with cause-effect paths and slots

• Modechart

- Graphical specification language based on concurrent finite state diagrams
- Constructs for the definition of event-driven real time systems:
  - modes
  - mode-transitions
  - events
  - constraints

- Mode can be either active or inactive
- State of a real time system is described by the collection of modes which are active
- Modes are hierarchically arranged
- Each mode can execute serially or in parallel, allowing to capture both concurrent and sequential behavior
- The semantic of modes and transitions is expressed in real time logic (RTL)

- Action is an operation which requires a non-zero but bounded amount of system resources (periodic or sporadic)
- Event marks a time value and requires no resources
- Behavior is captured by mode transitions
- Mode transitions are expressions which control the exit from one mode and entry into another

- Transitions
  - can be specified by timing constraints or
  - can be triggered by events in the system or
  - can be controlled by predicates on the behavior of modes
- Events include
  - mode-entry
  - mode-exit
  - transition
  - external

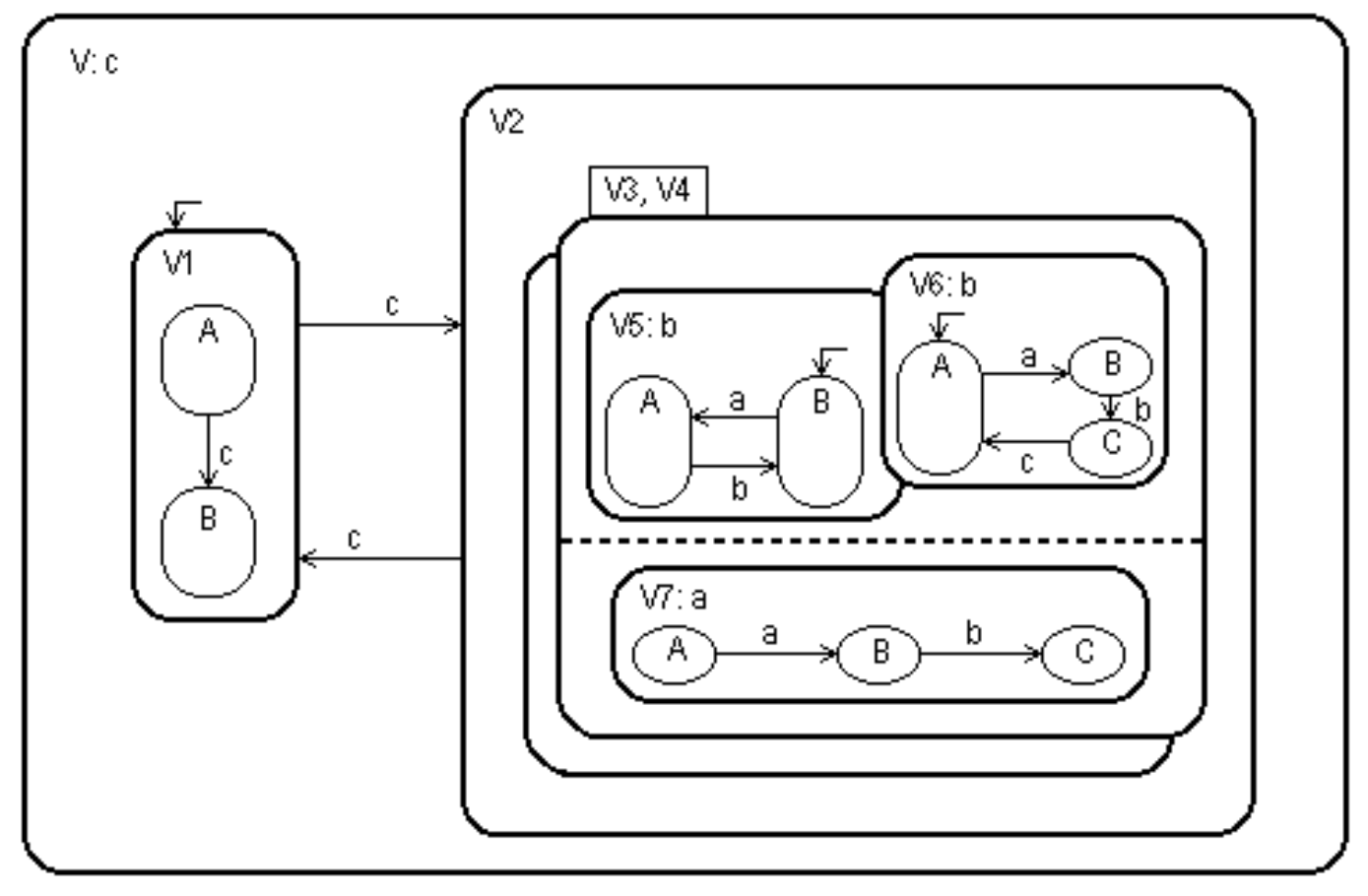
• Viewchart

- Statecharts is often criticized for
  - explosion of states
  - global name space
- Viewchart consists of a hierarchical composition of views
- Views are represented just like states
- Viewchart limits the scope of broadcast communications

- Composition of views requires communication between the views
- Viewcharts distinguish two types of events:
  - belong to (or are owned by) the view
  - do not belong to the view
- Composition of views requires communication between the views
- Action belongs to the view (or views) that generates (or generate) the action

- Composing behavioral views
  - **SEPARATE** composition of views
    - all the views are active
    - no transition between the views is allowed
    - the scopes of all the elements are unaffected
    - any subview or state in one view is hidden from (cannot be referenced by) the other views
  - **OR** composition of views
    - only one view can be active
    - there can be transitions between the views

- Composing behavioral views (cont'd)
  - **AND** composition of views
    - all the views are active
    - the scopes of all the elements owned by each view are extended to the other views
    - all the subviews and states in one view are visible to (can be referenced by) the other views

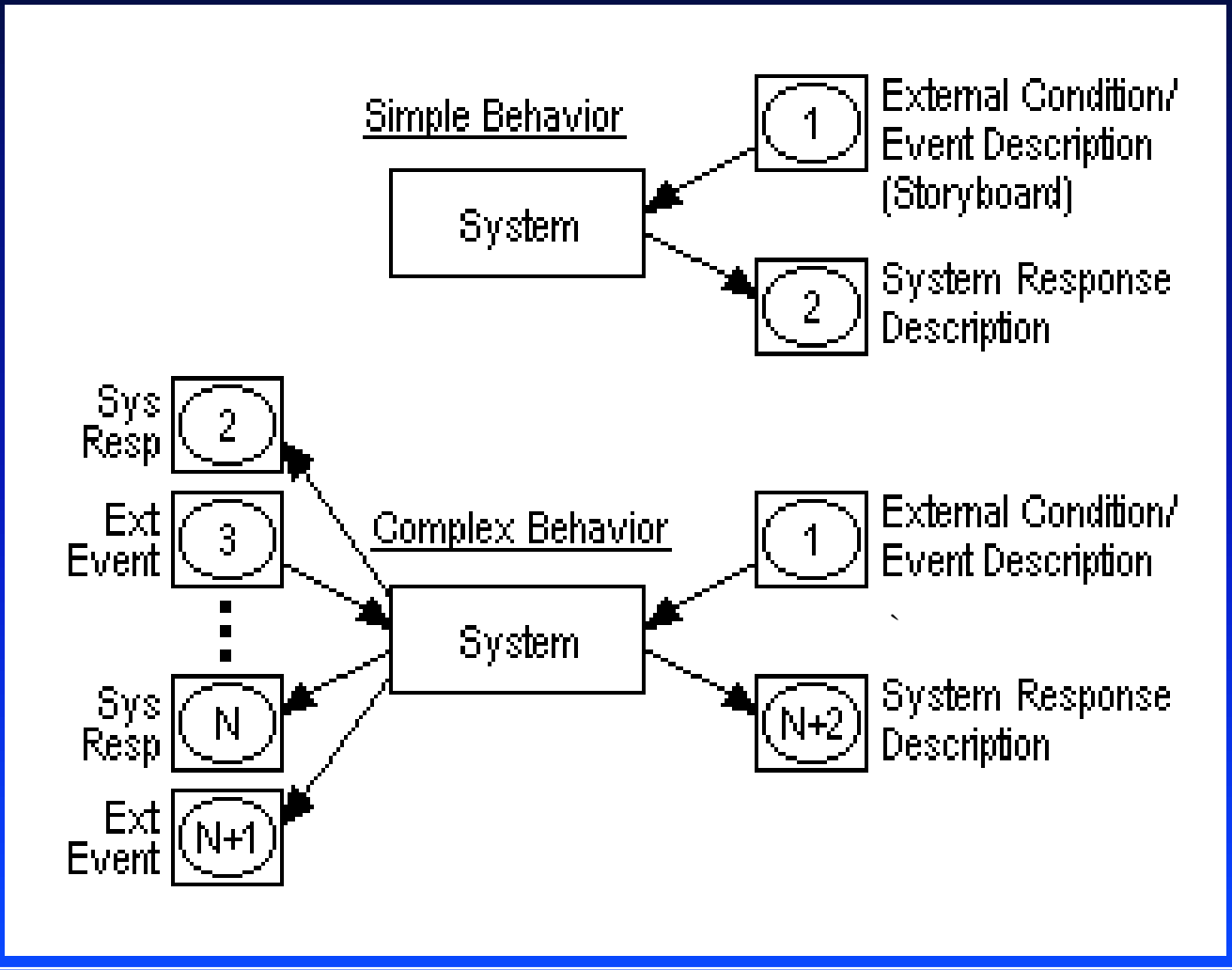

V: c
V2
V3, V4
V1
A
c
B
c
c
V5: b
A
a
B
b
V6: b
A
a
B
b
C
c
V7: a
A
a
B
b
C

• Design Capture Views

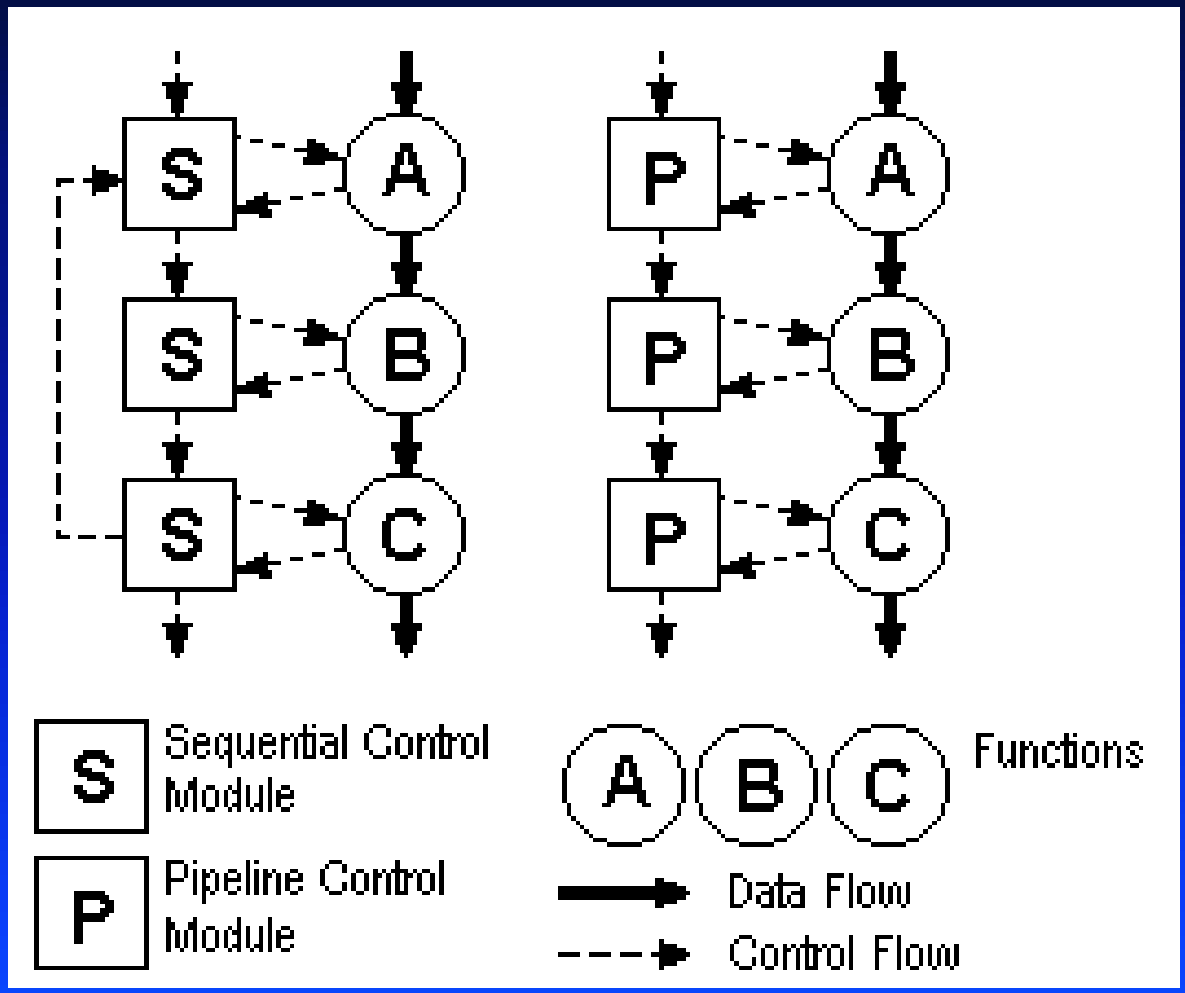
- Principal system design perspectives:
  - Environmental
  - Informational
  - Functional
  - Behavioral
  - Implementation

- Behavioral capture addresses three key aspects of representing system behavior:
  - response to stimuli as observed externally
    - system as a black box
  - r-t behavior **functional** design options
    - control, timing, and synchronization aspects of the system functions and data without regard to specific implementation in resources
  - r-t behavior **implementation** design options
    - response time, queue lengths, and other aspects of the system behavior as embodied in hardware, software and human resource and resource architectures

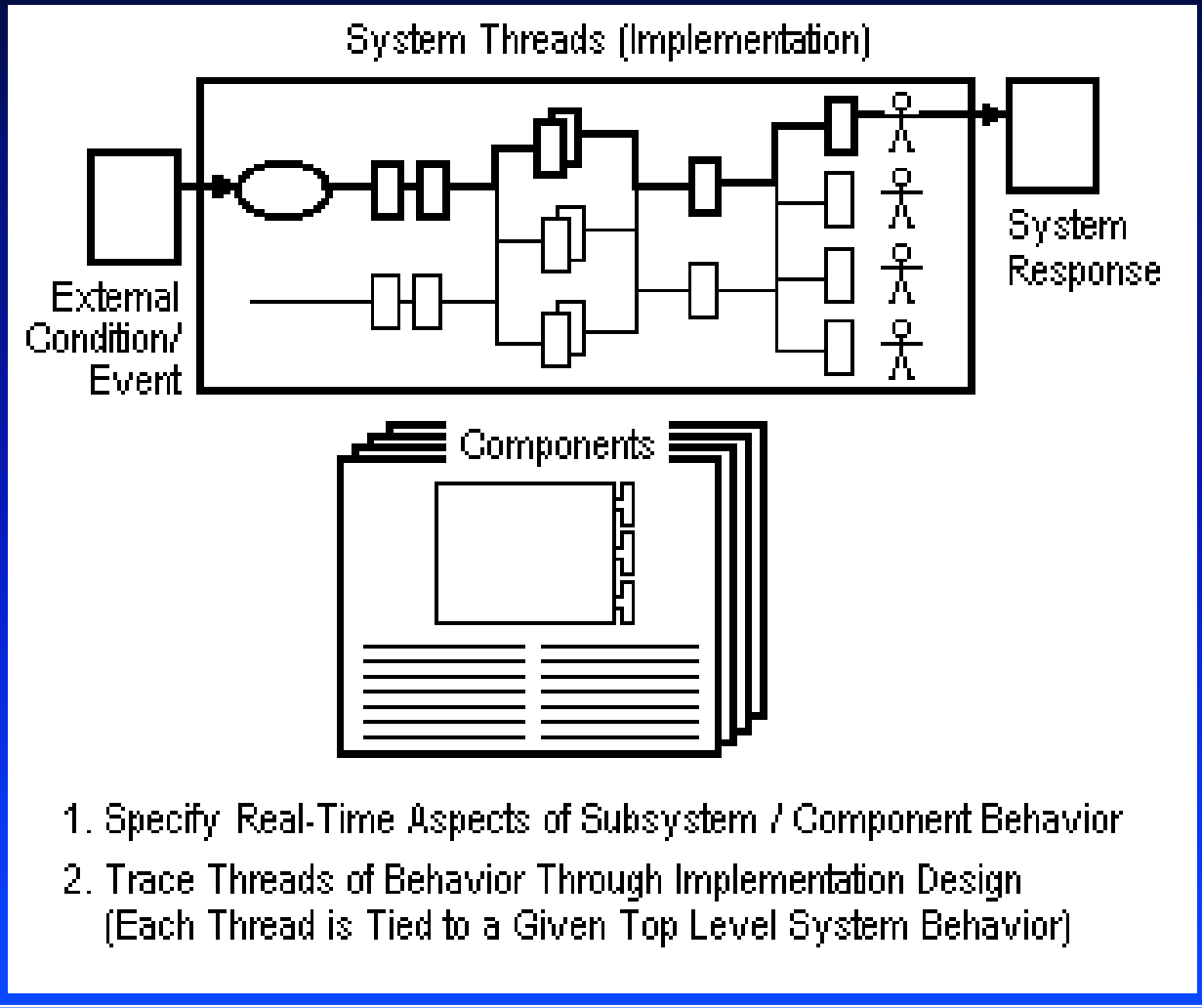
- response to stimuli as observed externally



- r-t behavior **functional** design options



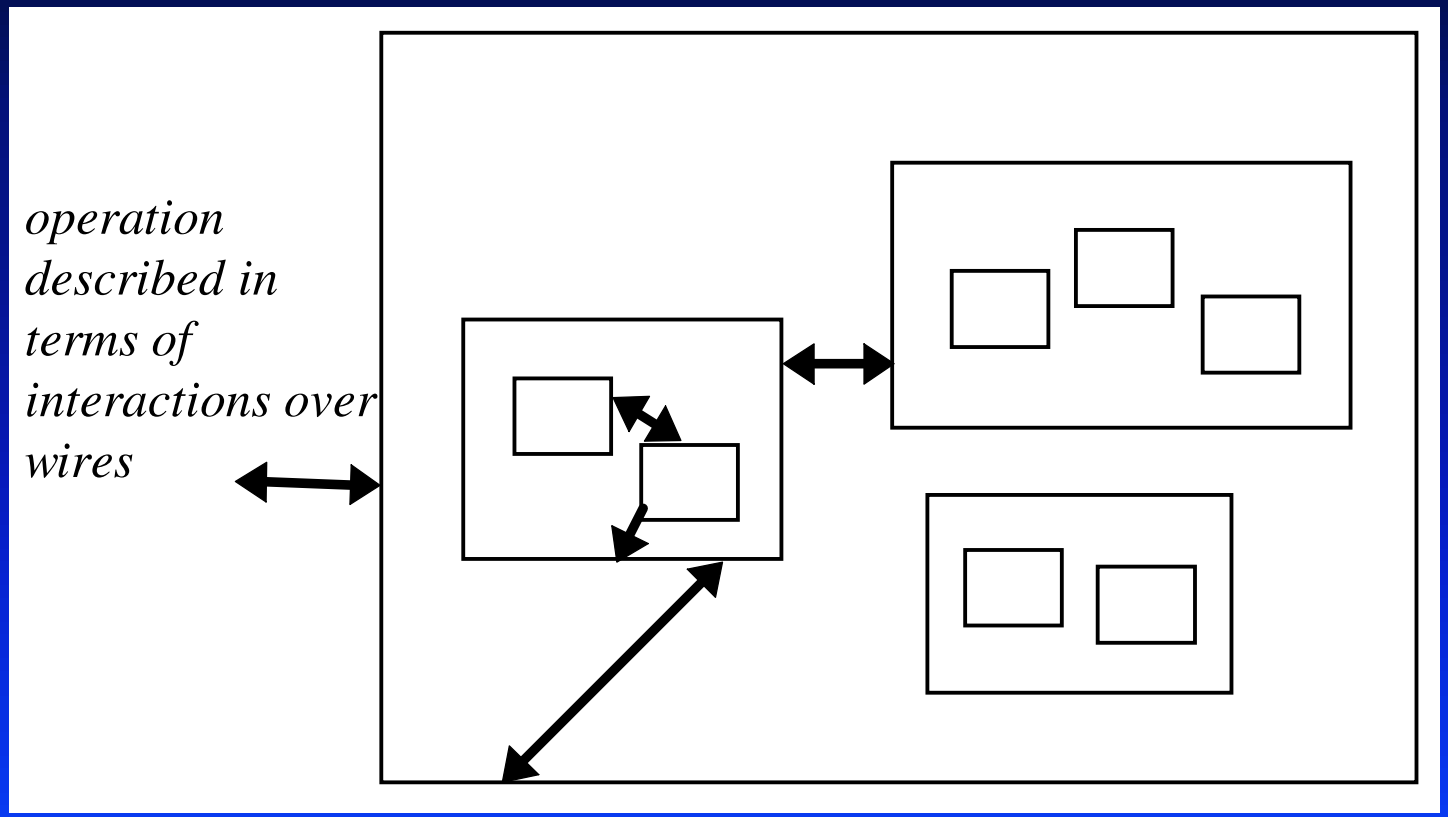
- r-t behavior **implementation** design options



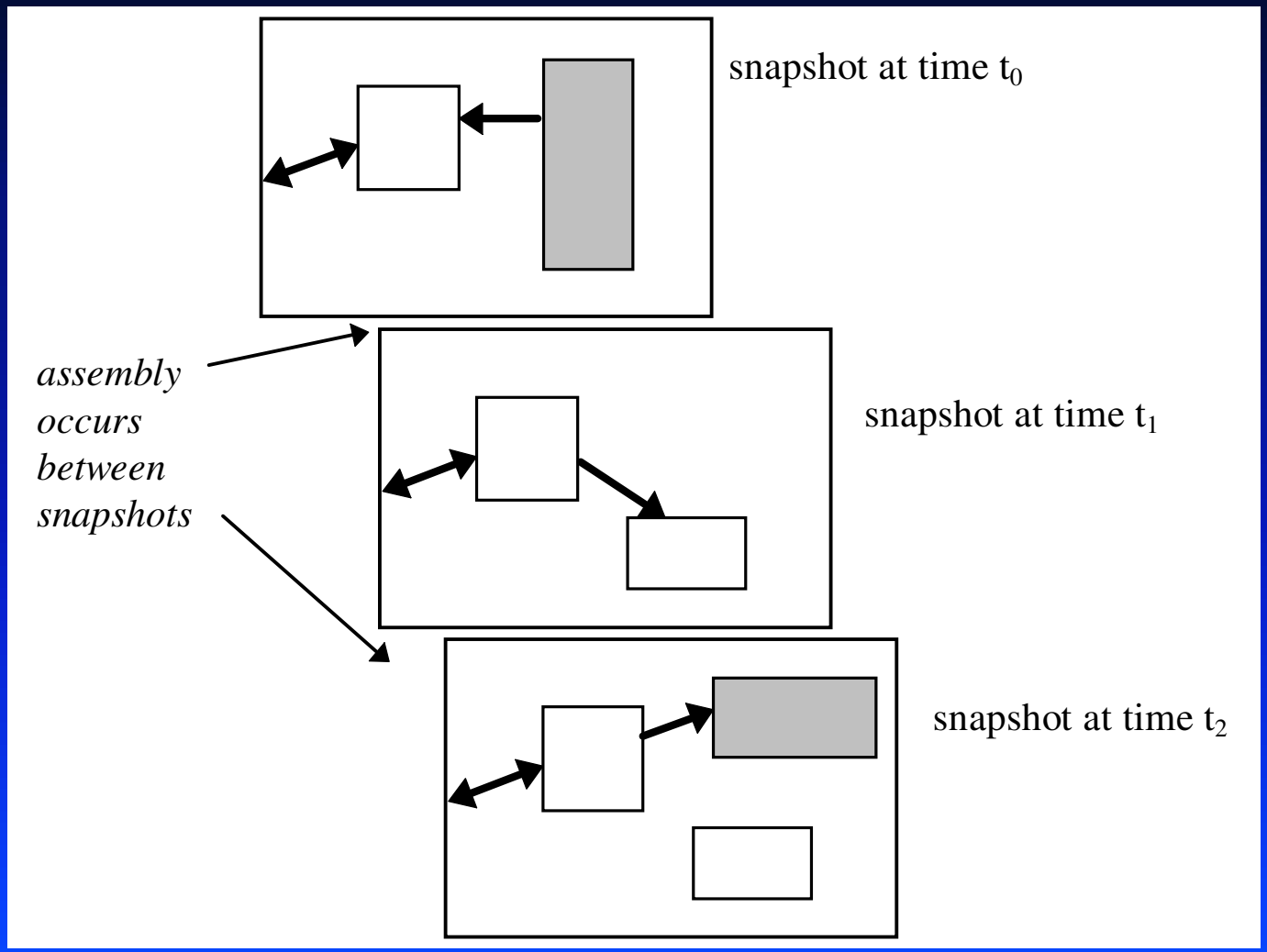
• Use Case Maps

- Systems characterized at a high level
  - context diagrams
  - connective tissue - intercomponent wires
- Complexity Factors in Systems
  - Operation
  - Assembly
  - Manufacturing

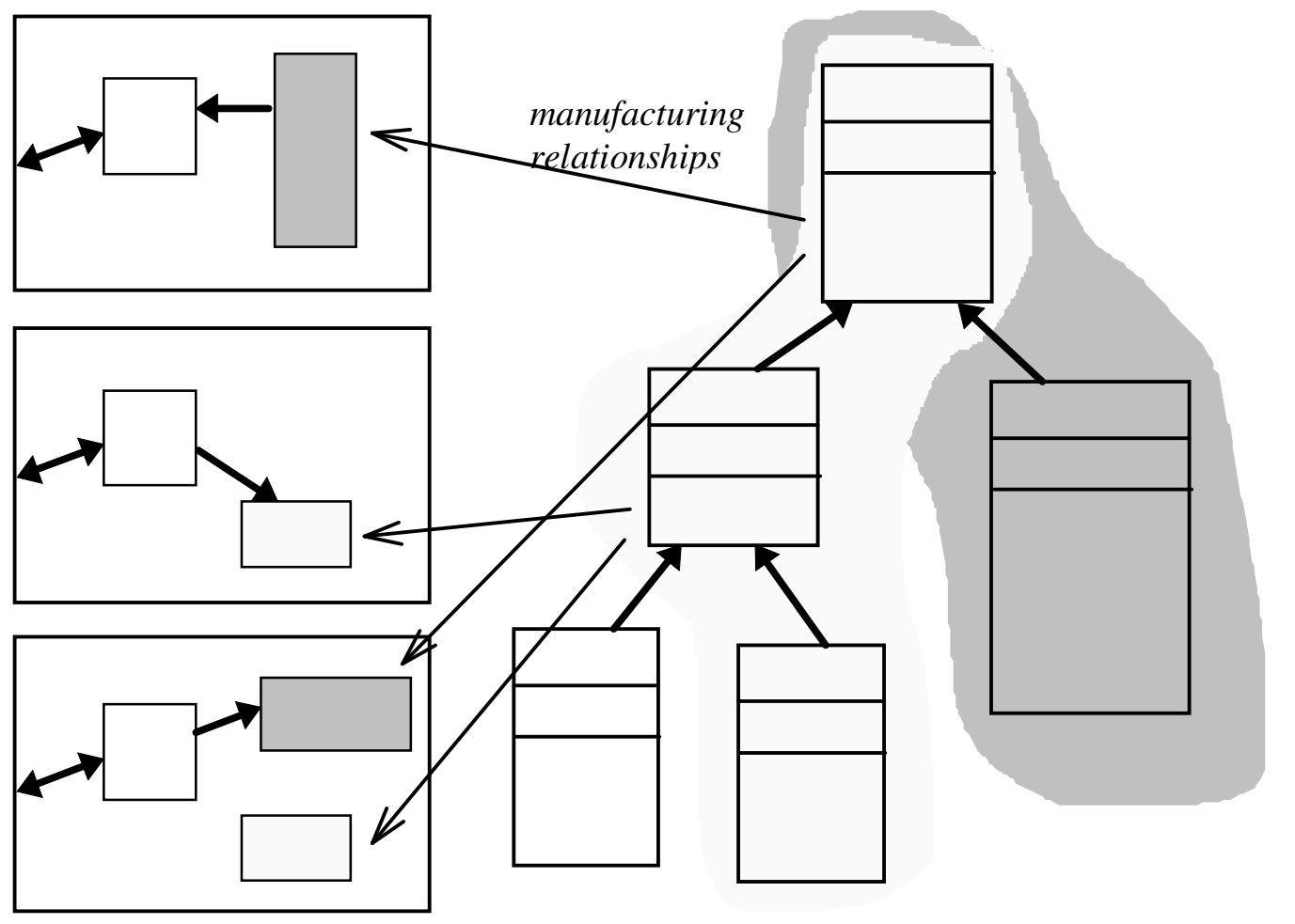
- Operation



Assembly


snapshot at time t_0
assembly occurs between snapshots
snapshot at time t_1
snapshot at time t_2

- Manufacturing



Essence of Use Case Maps

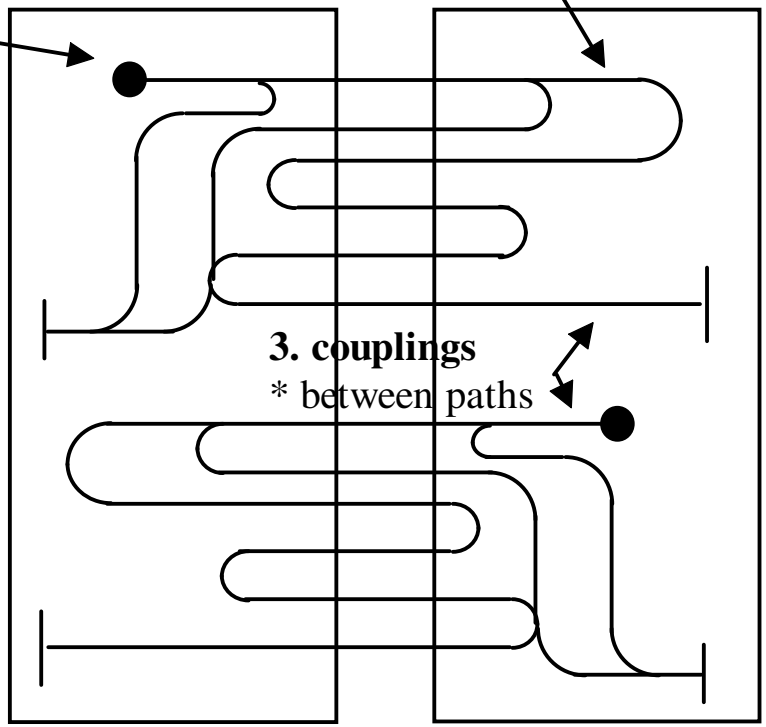
- Operation
  - Wires are replaced by cause-effect paths
  - Behavior is represented by cause-effect sequences
- Assembly
  - The modeling is eliminated
  - The changing components part is modeled through the use of slots
- Manufacturing
  - The modeling is eliminated
  - Use case maps may be developed independently of class descriptions

Use case maps have four elements

2. responsibilities
* labeled points on paths
* coarse-grained units of activity
* bound to components
* high-level view

1. paths
* of use case scenarios
* from stimulus to response
* cause-effect chains through system
* routes for slot occupants
* first class models

4. components
* characterized by responsibilities
* no internal logic
* no interfaces
* fixed & slots



3. couplings
* between paths

Modeling and Programming

- Real-Time ANSA
- ROOM - Real-Time O-O Modeling
- JAVA

- Real-Time ANSA

- The ANSA architecture
- Distributed Object Execution
- Designing a Real-Time Programming Model

• Real-Time ANSA (cont'd)

- A Real-Time Programming Model
  - ANSA object execution
  - Real-time objects
  - Real-time object invocation
  - Scheduling
  - Priority scheduling
  - Priority management and priority inheritance
  - Other scheduling paradigms

- The ANSA architecture
  - Based on distributed object (client/server) model
  - Services are provided by typed interfaces
  - Conformance relationship is defined to allow interface substitution
  - Interfaces contain operations which are invoked to make use of the service offered by that interface

Distributed Object Execution

- Execution models are required to address how
  - the computational activities are carried out
  - shared resources are used (the manner in which contention for system resources is resolved taking into account timing constraints of real-time activities)
- Distributed real-time systems must provide-support for the specialized requirements of real-time communication, tasking, scheduling, and control

Designing a Real-Time Programming Model

- **Abstractions**, **mechanisms** and **policies** are developed to allow a programmer to access and control the resources. Tasks (processor resources) and communication-channels (communication resources) are considered the most important system resources. Both static and dynamic resource allocation are supported
- **Predictability**, **user control** and **mission criticality** are the main concerns

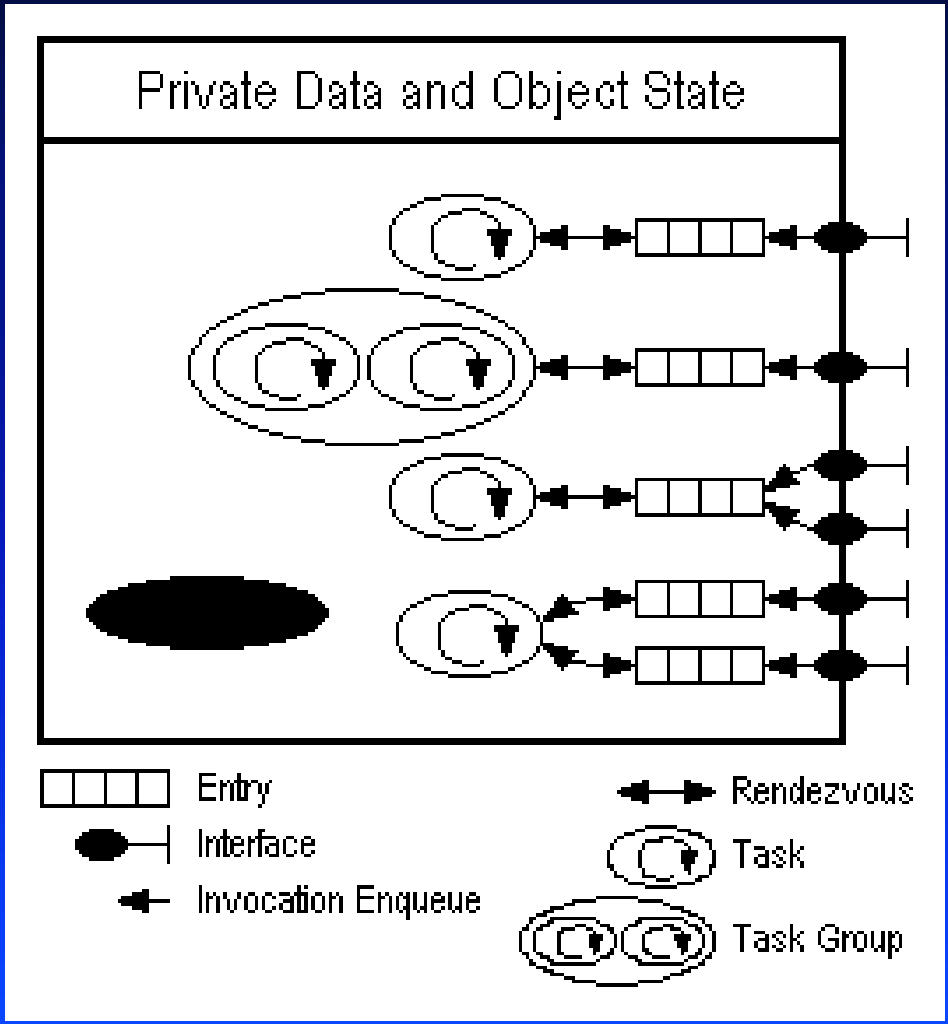
A Real-Time Programming Model

- ANSA object execution
  - objects export services through interfaces
  - threads are created
    - explicitly for concurrent activities
    - implicitly by the invocations between objects. (thread is a distinct run-time agent for a client)
  - the infrastructure (capsule) is in charge of the management of resources (tasks , buffers etc.) , and of their allocation to the different threads
  - system behavior is completely dependent on the resource management policy, totally non-deterministic, and nothing can be guaranteed; it depends entirely on the system workload

A Real-Time Programming Model (cont'd)

- Real-Time objects
  - Entry is a thread queue with a record of control data.
  - Entry may be created dynamically, and interfaces of an object may be bound to it. Each operation request on the interface will be transferred (by the infrastructure) to a thread enqueued on the entry. Any thread representing a computational activity is also spawned on an entry
  - Thread is a sequence of instructions representing a unit of potentially concurrent activity. Task is a virtual processor which provides a thread with the resources it requires to progress

A Real-Time Programming Model (cont'd)



A Real-Time Programming Model (cont'd)

- Real-time object invocation
  - Invocation is conveyed as a message to the invoked object, and is then transferred to a thread in the capsule where the invoked object resides
  - Urgency information is used by allowing the association of an optional priority and/or deadline with each invocation. This parameter becomes a property of the thread, which may then be used as a scheduling parameter on the server site
  - Priority and/or deadline is just a client's view, the result is also determined by the scheduling policy and the resources allocated for the service

A Real-Time Programming Model (cont'd)

Scheduling

The main goal is to allow the maximum control of scheduling at the application level

Thread scheduling.

- How a thread is enqueued in an entry. Policy may be either system-defined or application provided one
- How the thread scheduling parameters are used by the task. This is defined by a task/thread rendezvous policy

Nucleus scheduling

- How the real processor(s) is assigned to tasks. Preemption is used together with task scheduling parameters to order (either partially or completely) the otherwise non-deterministic behavior of the task execution

A Real-Time Programming Model (cont'd)

- Priority scheduling
  - Priority and scheduling are different issues
  - The priority queuing of threads in an entry is incorporated to support such a view of real-time applications. While priority is a well defined and generally applicable notion, its role in task scheduling needs to be carefully examined. A clear definition of the priority inheritance and priority ceiling - used when the enforced synchronization during a task and a thread rendezvous - is needed to understand how priority works on tasking

A Real-Time Programming Model (cont'd)

- Priority management and priority inheritance
  - task's static priority (that declared in its creation)
  - task's dynamic priority (static value enhanced by a rendezvous or an explicit change of priority)
  - Two-level priority inheritance scheme defined.
    - Priority inheritance
    - Transitive priority inheritance
- Other scheduling paradigms
  - deadline based thread scheduling
  - earliest deadline first preemptive scheduling service

• ROOM - Real-Time O-O Modeling

- The ROOM Language
- The Communication Model
- Modeling Structure
- Modeling Behavior

The ROOM Language

- Primary aspects of a software architecture
  - **Structural** aspects deals with the high-level decomposition of a system into components and their usage relationships
  - **Behavioral** aspect covers the dynamics of the system; i.e., the high-level interactions between architectural components required to achieve the desired functionality
- Architectural genericity can be used to
  - define product families
  - control system evolution ensuring that the initial high-level system requirements remain intact

The ROOM Language (cont'd)

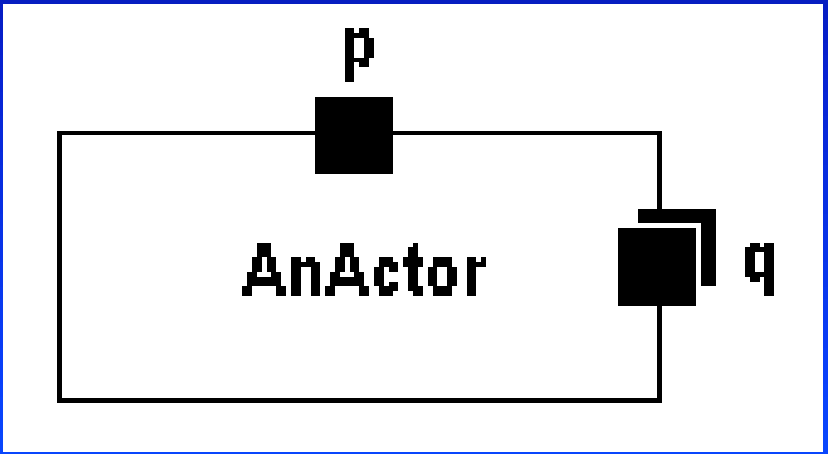
- ROOM overcomes the architectural decay by defining a system specification across two distinct but formally correlated levels:
  - ROOM uses a graphical syntax at the **Schematic Level** supplemented by formal textual annotations
  - The **Detail Level** language addresses detail that a typical implementation language covers, An existing language, such as C++ can serve this purpose

The Communication Model

- Event is signaled by the arrival of a message
- A **protocol class** defines a reusable message protocol specification. It consists of
  - a list of message types (<signal, data class> pairs)
  - individual directions (incoming or outgoing)
  - an optional message exchange sequences specification
- Specifying protocols as classes provides the ability to define abstract protocols that can be specialized in different ways through subclassing

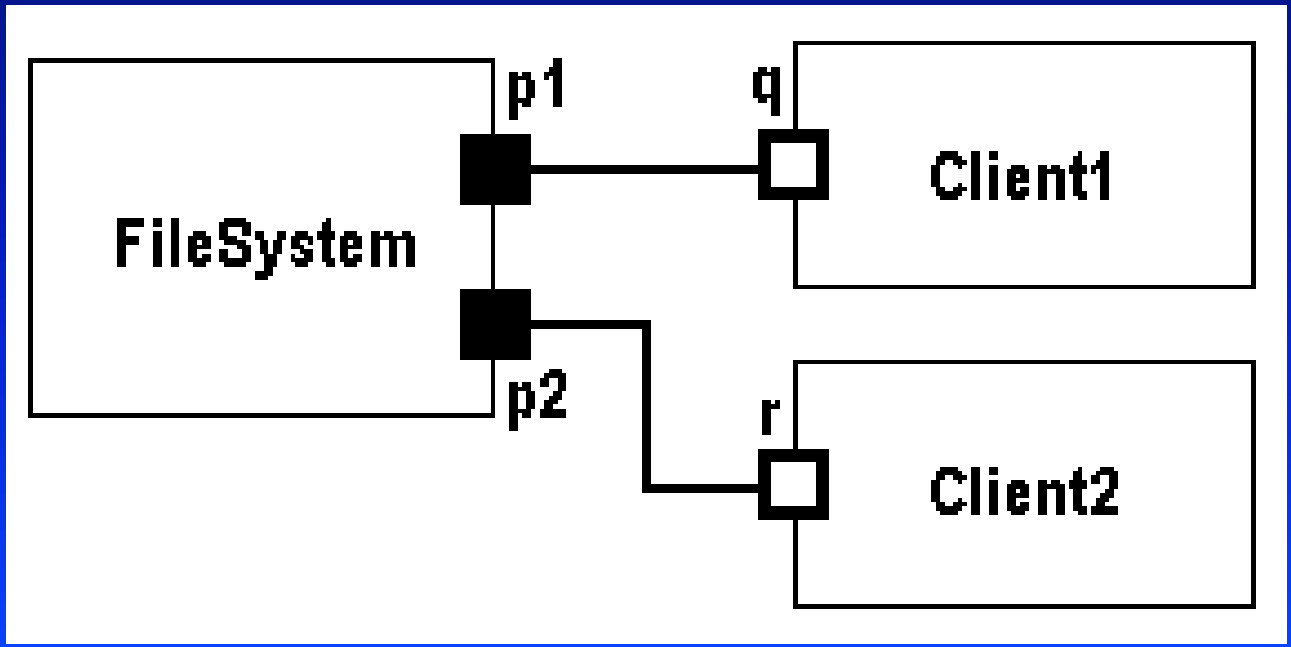
Modeling Structure

- **Actor** (instance of an actor class) is a concurrent object (it can operate in parallel with other actors) responsible for performing some specific function
- Actors communicate through **Port**s (instance of a protocol class)



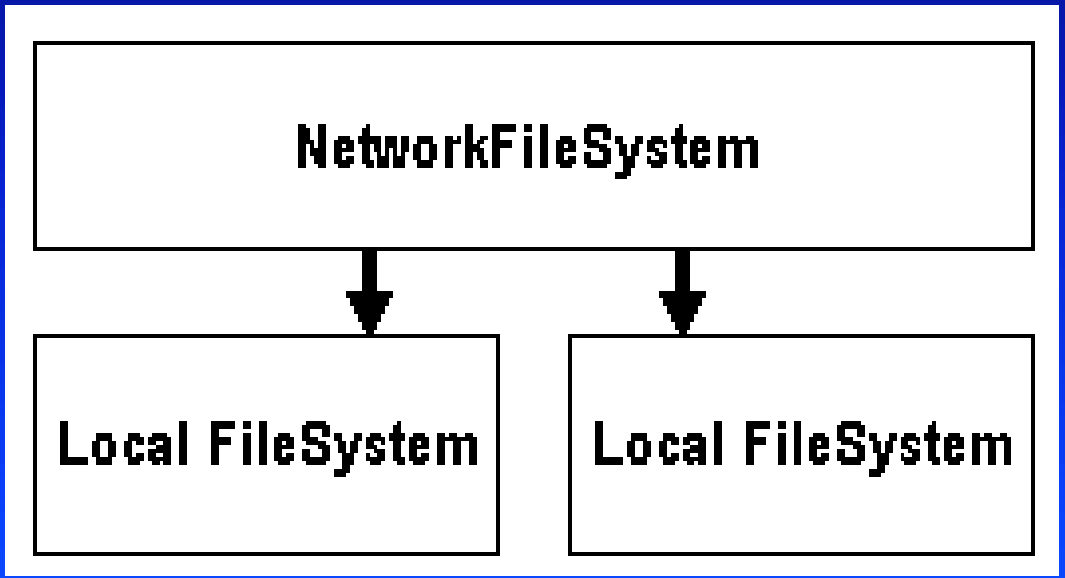
Modeling Structure (cont'd)

- composition mechanisms:
  - **Binding** models a communication channel that connects two ports



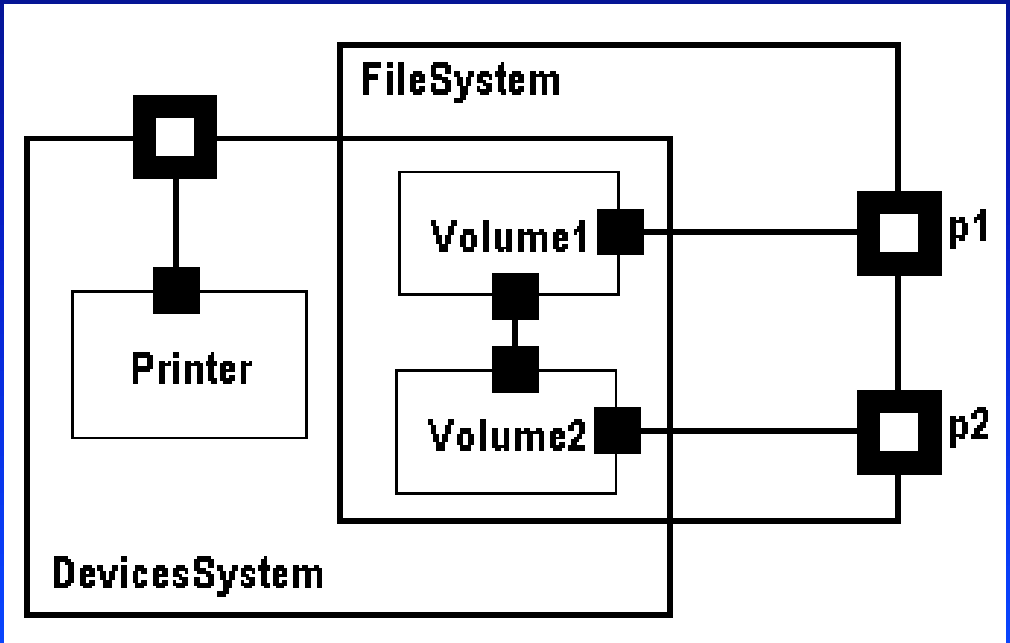
Modeling Structure (cont'd)

- composition mechanisms:
  - **Layer connection** connects one or more service access points (SAP) on the client to a service provision point (SPP) on the server
  - SAPs and SPPs are also instances of protocol classes



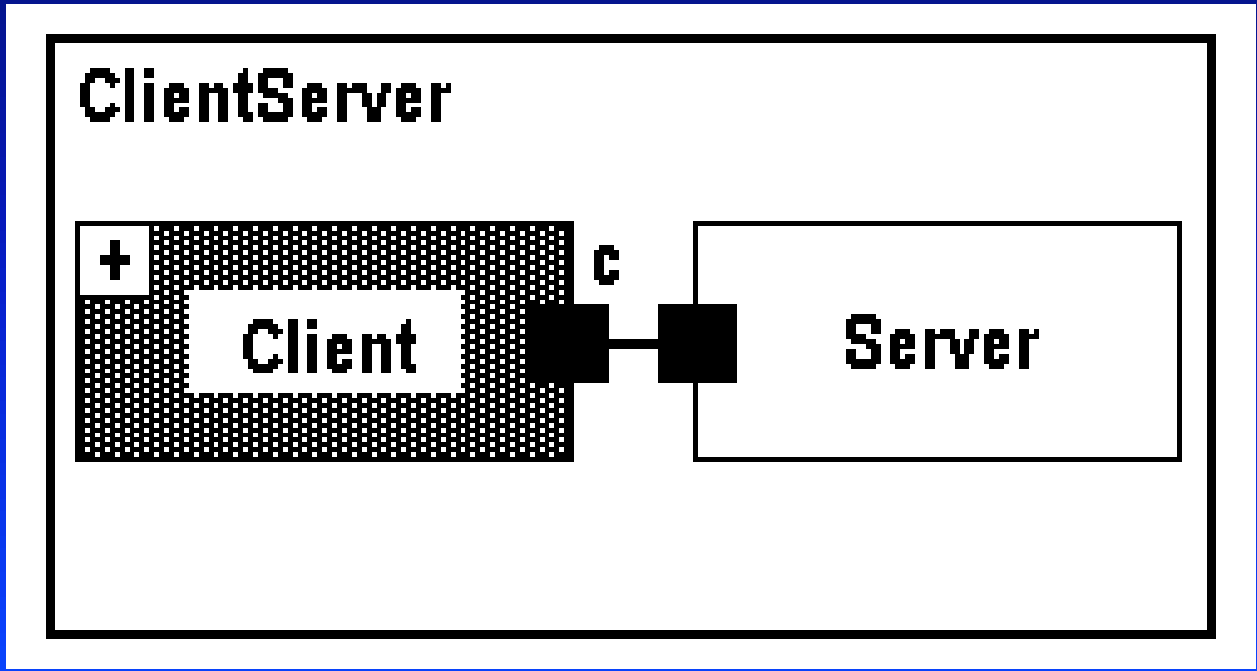
Modeling Structure (cont'd)

- composition mechanisms:
  - **containment** = compositionality relationships
  - multiple containment - some actors may simultaneously be in more than one container



Modeling Structure (cont'd)

- transient relationships
  - place-holder components are called **imported actors**



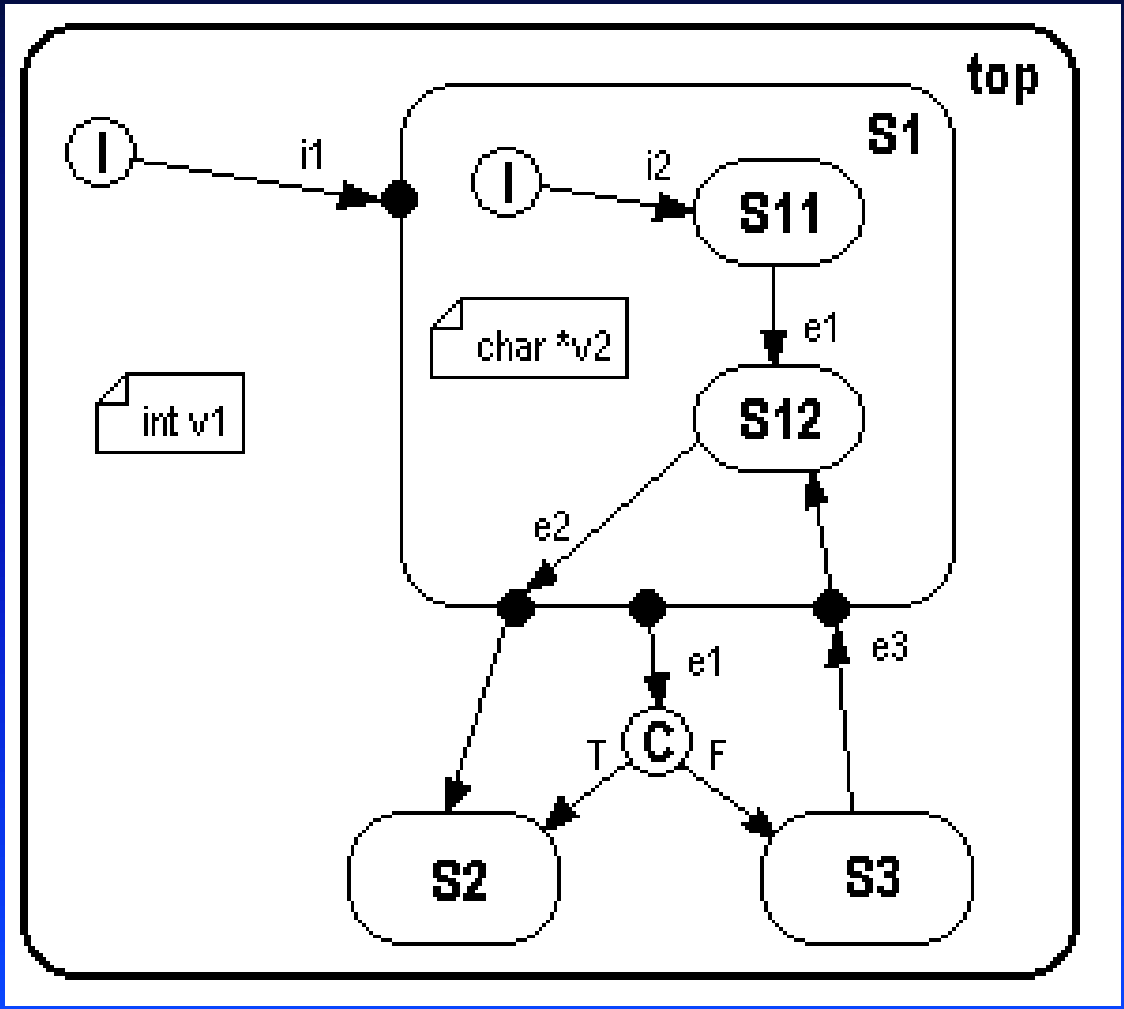
Modeling Behavior

- Behaviors of actors are the ultimate source and sink of all messages
- Structural and behavioral dimensions meet is at the interfaces of an actor (ports, SAPs, and SPPs)
- Behavior interfaces also serve as the "crossover" points between the Schematic Level and the Detail Level
- Hierarchical finite state machines called ROOMcharts

Modeling Behavior (cont'd)

- Event is represented by the arrival of a message at an actor interface. Depending on the current state of the behavior, the type of signal in the message, and the interface on which it arrives, the event will trigger a transition that leads to a change of state of the behavior
- ROOM uses a run-to-completion model of event processing
- Hierarchy in ROOMcharts means that a state can contain a state machine

Modeling Behavior (cont'd)



• Real-Time JAVA

- Java Background
- Java Overhead
- Real-Time Challenges
- Theoretical Foundations

• Real-Time JAVA (cont'd)

- Structure of a R-T Java Application
  - Loading of a Real-Time Activity
  - Configuration
  - Resource Negotiation
  - Execution of Real-Time Tasks
  - Blocking

Java Background

- Java provides standardized support for
  - multiple threads (lightweight tasks) and
  - automatic garbage collection
- Java execution model
  - Java programs are first translated into a fully portable standard byte code representation
  - Byte code is then available for execution on any Java virtual machine, (interpreter)
- Java virtual machine includes a Java byte-code analyzer that verifies the code does not contain dangerous requests

Java Overhead

- Garbage collection costs
  - Typical R-T Java programs spend about 30 % of their execution time performing garbage collection
  - Causes of higher overhead:
    - garbage collection effort is divided into many very small incremental steps
    - the garbage collector recycles memory much more aggressively than is typical of non-real-time systems
- Dynamic binding of method invocations costs
  - Performance cost of dynamic method dispatch in comparison with C is a minimum of 20% of execution time

Real-Time Challenges

- Java applications are highly dynamic. New r-t workloads arrive continually and must be integrated into the existing workload. This requires
  - dynamic management of memory
  - on-the-fly schedulability analysis
- Additional complications are:
  - Developer is not able to test the software in each environment in which it is expected to run
  - Each execution environment is likely to have a different mix of competing applications with which this application code must contend for CPU and memory resources

- Theoretical Foundations
  - Typical execution time of most code is much shorter than the worst-case execution time
  - Application developers are the best qualified to select function parameterizations that represent typical and/or worst-case invocations
  - Given task execution times and frequencies, rate-monotonic analysis allows quick and straightforward determination of whether a particular combination of real-time workloads is capable of running reliably within real-time constraints

Theoretical Foundations (cont'd)

- Static cyclic schedules offer a lower run-time overhead and fewer context switches than fully dynamic scheduling. As a possible run-time optimization, static cyclic dispatch tables may be created by the real-time executive. Rate-monotonic techniques offer a straightforward approach toward creation of static schedules
- The dynamic memory manager enforces a memory allocation budget for each real-time activity to prevent rogue tasks from interfering with the memory needs of other tasks

Structure of a R-T Java Application

Loading of a Real-Time Activity

- When a real-time activity is loaded, the byte-code analyzer performs special analysis
  - It builds a call graph for each of the real-time tasks that comprise the activity
  - For each of the methods the analyzer examines exception handlers and determine their worst-case execution times
- When a task is aborted, its pending exception handlers are allowed to execute. This gives the task an opportunity to release any mutual exclusion locks that it might hold and to perform other cleanup operations as necessary

Structure of a R-T Java Application (cont'd)

- Configuration
  - Configuration manager is responsible for determining
    - The worst-case time required to execute particular segments of code
    - The typical time required to execute particular code segments
    - The amount of memory required to represent particular data structures in the local environment
  - Configuration manager returns to the real-time executive a representation of the activity's resource needs in terms of the following quantities:

Structure of a R-T Java Application (cont'd)

Configuration (cont'd)

- The minimum total amount of live memory
- The desired total amount of live memory
- The minimum acceptable rate for allocation of new memory (which is the rate at which the real-time garbage collector must be able to recycle dead objects)
- The desired rate for allocation of new memory
- For each of the real-time tasks that comprise this real-time activity:
  - The period of execution
  - The minimum amount of execution time / period
  - The desired amount of execution time / period
  - The smallest acceptable time slice size

Structure of a R-T Java Application (cont'd)

- Resource Negotiation
  - Resource negotiation itself is not a real-time activity. While resource negotiation is taking place, the system continues to run the previously negotiated real-time workload
  - Real-time executive
    - computes CPU-time and memory budgets for each activity
    - offers time and memory budgets to each real-time activity
  - Activity accepts or rejects the proposed budget
  - If any activity rejects its proposed budget, the real-time executive may endeavor to propose an alternative more acceptable budget at its discretion

Structure of a R-T Java Application (cont'd)

- Resource Negotiation (cont'd)
  - If, the real-time executive decides that it is unable to accommodate the new activity, it simply reports to the application that attempted to introduce this new activity into the environment that it was unsuccessful in its attempt
  - This application then has the option of
    - removing other real-time activities from the system
    - requesting that certain activities be given preferential treatment during resource negotiation
    - resubmitting its request to add the activity to the current workload

Structure of a R-T Java Application (cont'd)

- Execution of Real-Time Tasks
  - Min. periodic execution time at least enough to execute startup and finalization code segments
  - The work component may include unbounded loops and recursion (object libraries ?)
  - The real-time executive
    - allows the work component of each task to run for as much of the task's requested ideal execution time as is available
    - aborts the task when the available time is exhausted, always leaving sufficient time to execute the work component's pending exception handlers followed by the task's finalization code

Structure of a R-T Java Application (cont'd)

- Blocking
  - On a single processor Java virtual machine, the implementation of atomic statements is straightforward. If there is sufficient time in the current time slot to execute the atomic statement, we do so. Otherwise, we suspend the task until its next scheduled time slot.
  - With multiprocessor implementations, additional run-time overhead may be required to prevent interleaved execution of atomic statements. But the general philosophy is the same: No task is allowed to hold a lock during times when the task is not executing.