Object-oriented Design of Real-time Systems

Purpose

- **Architecture and behaviour**
  - **architecture families and issues**
- **Specifying behaviour**
  - **at the system level (use cases, collaborations, Use Case Maps)**
  - **at the module level (state machines)**
  - **developing module specs from system requirements**
- **Concurrency and architecture**
  - **developing specifications**

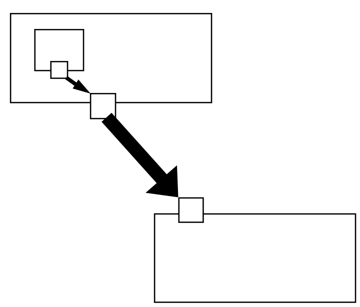
Types of real-time systems

- reactive to continuous signals.... for instance
  - signal processing (speech and radar processing)
  - control systems (aircraft, cruise control, factory automation)
- reactive to discrete events.... for instance
  - elevator control, telephony
- data request handling.... for instance
  - web-based systems (e-commerce)
  - transaction processing (point-of-sale, banking)
  - client-server systems

Typical architectural/behavioral design solutions

- in each of these domains we shall study commonly-used patterns for the architecture and behavior of designs
  - nature of solution
  - strengths, when to apply it
- we shall also consider extensions and alternatives
  - additional concurrency in asynchronous designs
  - replication

Elements of architecture



- *components* (may be *nested*)
  - object
  - module
  - task or concurrent process
- *interactions* or connectors
  - association
  - ports and protocols
    - synchronous message (OO message, or procedure call, or RPC)
    - async message (no reply)
    - sync/async combinations and variations (e.g. async RPC)
    - conversations

ADLs and ROSE-RT tool

(Architecture description language, e.g. Aesop, Newton, ACME)

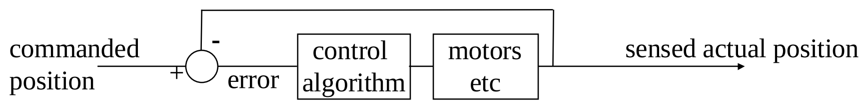
- ROSE-RT uses UML (OMG standard) to define architecture:
  - UML classes and associations to define components and interactions
  - UML aggregation to show nested components
  - R-RT *protocol* to define the nature of the interaction
- switches to a special CAPSULE model to define behaviour
  - a *capsule* is a component, (== class)
    - all capsules are potentially concurrent
  - a *port* is the attachment of a protocol/interaction
  - a *state machine* defines behavior of the capsule
    - reacts to events on ports, puts out events, computes

ADLs and SDL (TAU tool)

- SDL = Specification and Description Language
  - ITU standard
  - linked with Message Sequence charts, also an ITU standard
- components are *processes*
- connections are *connections* with *signal-lists*
- all interactions are async
  - synchronous interactions can be built up from async

Typical reactive-continuous systems

- clock-driven, to convert the continuous position, temperature or other signal into a series of values
  - e.g. sample the aircraft elevator position, drive the motors that change its position



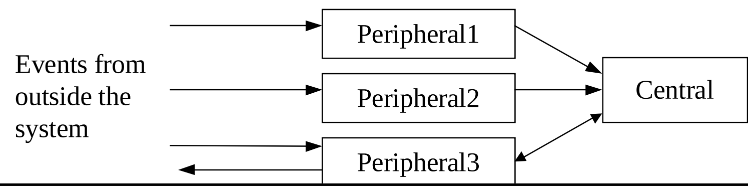
  - sampling, must be fast enough (twice the max frequency)
  - process the sample, put out the output
- possibly several independent sampling-processing paths, for multiple control loops
  - independent programs, each driven by a clock

Reactive-continuous system software architecture

- separate concurrent tasks, each responding to its own clock
  - sometimes several respond to the same clock
  - deadline to complete processing before the next clock event
- supervisory control is separate, logging conditions via stored data, and setting parameters.
  - heirarchy of subsystem monitors in complex systems
  - operator intervention via terminal interface transactions

Typical reactive-discrete systems

- events come from the outside, to signal completion of external activities or changes of state in the external "plant"
  - interrupts, at unpredictable times
  - e.g. elevator arrives at floor; door open; user button-push
  - e.g. telephone off-hook, on-hook
- typically there is central control to integrate all the event-handling and make a coherent operation of the system. (May be partitioned)
  - there may also be peripheral tasks to capture the events and even to handle some aspects of them (local loop)



Reactive-discrete system software architecture

- "central" control is sometimes distributed, as in telephony
  - originating terminal control
  - path points
  - destination terminal control
- heirarchical division of concerns and processing may occur in complex systems
  - truly central concerns (routing of calls)
  - truly peripheral concerns (eg, ring tone or dial tone generation)
  - intermediate levels (control of progress of call, handling call waiting or call hold).
- hard or soft real-time requirements
  - hard... all responses must complete by their deadline
  - soft... some percentage must complete by their deadline

Mixed continuous and discrete systems

- control systems often also have event reaction subsystems
  - e.g. a fire alarm or a passenger button-push, on an aircraft
- similarly a primarily event-driven system may have clock events that cause conditions to be polled
- clocked processing may raise exceptions that are then unpredictable discrete events for further processing

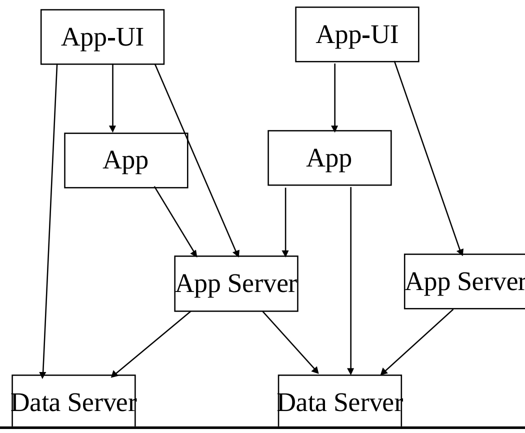
Typical data request handling systems

- essential unit of processing is a transaction
  - a “strong” transaction has ACID properties
    - atomicity (parts are indivisible)
    - consistency (satisfies some properties that prevent errors)
    - isolation (equivalent to serial execution)
    - durability (crash-resistant)
  - a “weak” transaction is a request for processing, without so many guarantees (e.g. a web page request).
- common factor is a user request, with data, and with multiple possible processing paths.
- usually has soft real-time requirements (e.g. 95% of responses complete within T sec). (Hard-real-time requirements exist too)

94.586 Object-Oriented Design of Real-time Systems...Winter 2001 A-13

Request-handling system architecture

- often client-server, including multi-tier CS....
  - Call return RPC infrastructure



94.586 Object-Oriented Design of Real-time Systems...Winter 2001 A-14

Request-handling: Banking system... Use Cases

- simple ATM system to execute a single transaction:
  - validate PIN, do operation
  - Gomaa example.....

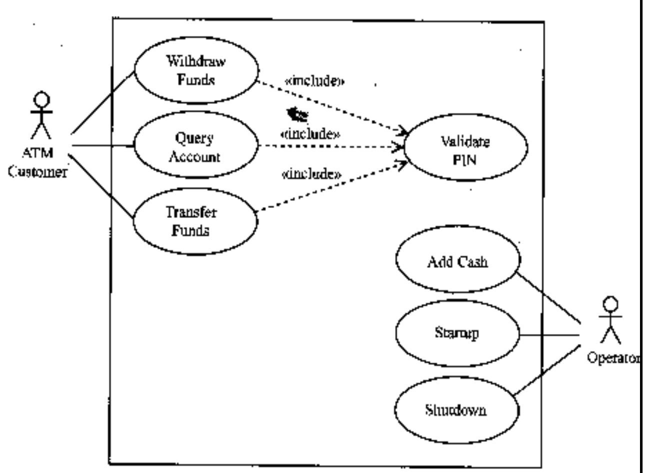


Figure 19.1 *Banking System use case model*

ATM... unbound state machine for behaviour

- states are system states
  - waiting for an event
  - processing
- transitions are reactions to events or to result of processing
- "validate PIN" Use Case as state machine...

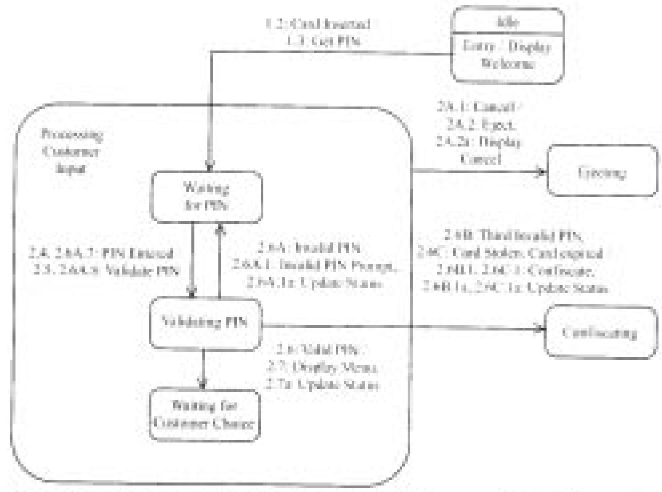
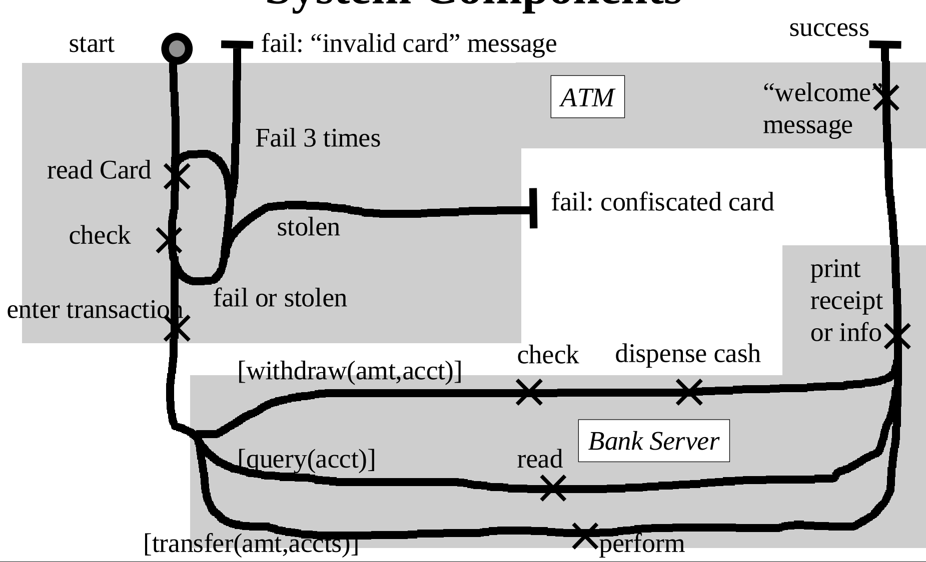


Figure 11.11 *Statechart for ATM Control: Validate PIN use case, showing alternatives*

Use Case Map shows scenarios with Major System Components



ATM side... Details: collaboration diagram

- roles and interactions
- controller
- devices:
  - card reader
  - cash dispenser
  - printer

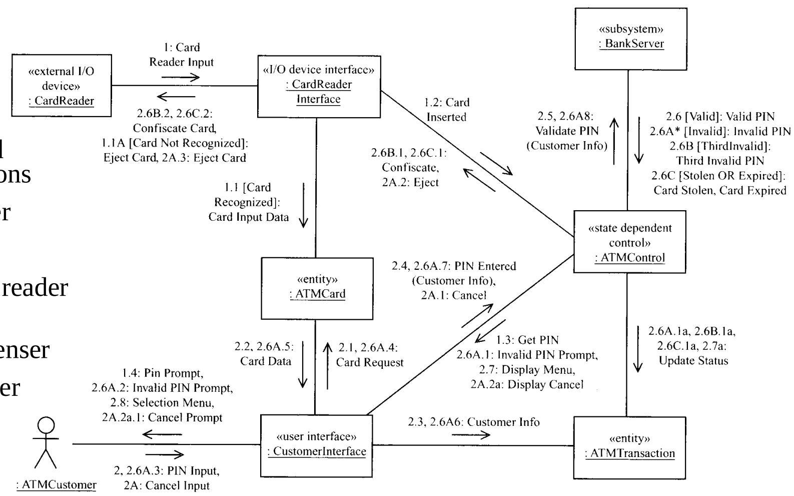


Figure 11.10 *Collaboration diagram: Validate PIN use case, generic form showing alternatives*

Bank Server side... Details: collab diag

- one concurrent task at the server

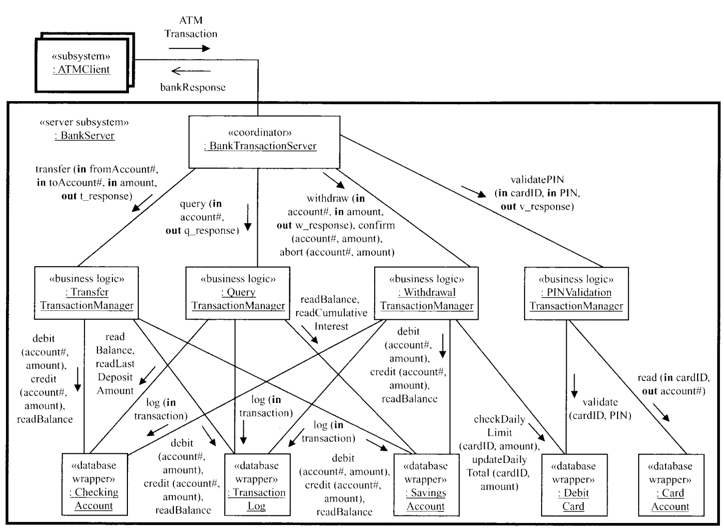


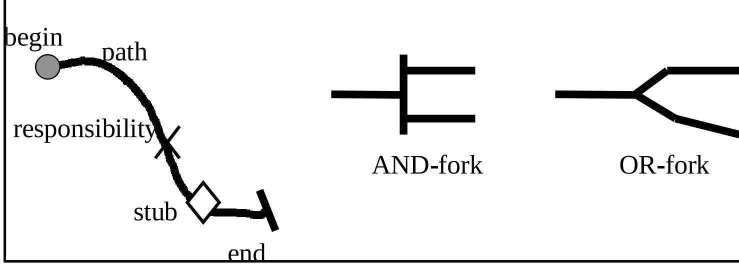
Figure 19.37 *Revised concurrent collaboration diagram for Bank Server Subsystem*

System behaviour specification... overview

- Use cases describe the operations of the system, as seen by actors outside the system
  - multiple use cases
  - broken down into components (sub-use cases) which may be shared
  - inheritance for specializations
- Use Case Map shows several Use Cases and the collaboration of components
- UML collaboration
  - an inside view of a single scenario
  - a sequence of operations by conceptual, planned or concrete objects
  - an object in a collaboration can stand for a subsystem
- UML sequence diagram
  - time-oriented view of the same collaboration, emphasizing messages
  - can also show parallelism
  - a limited special version of the ITU Message Sequence Chart

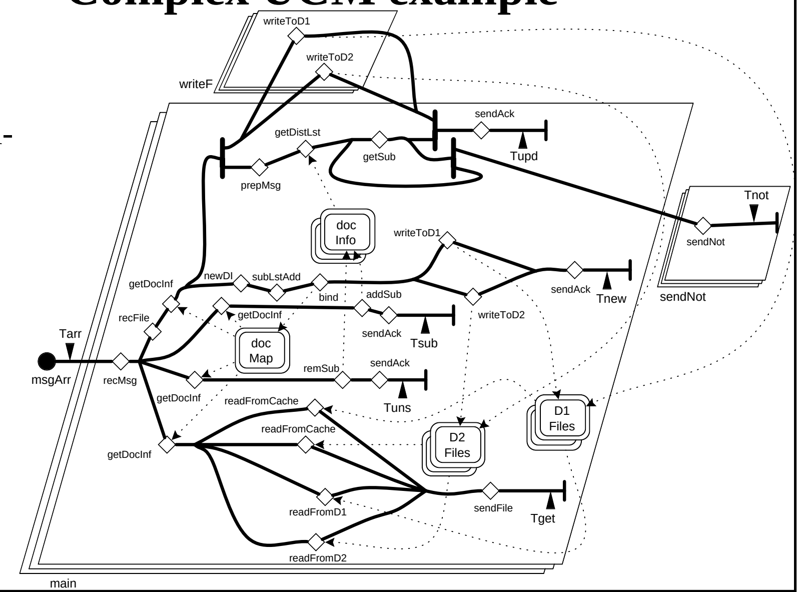
Behaviour specification by Use Case Maps

- unbound maps specify sequences of operations or *responsibilities*
- a path == a line over a diagram, time runs along it
- parallelism and alternatives are easy to show
- multiple scenarios are easy to show
- complex diagrams managed by stubs to hide detail



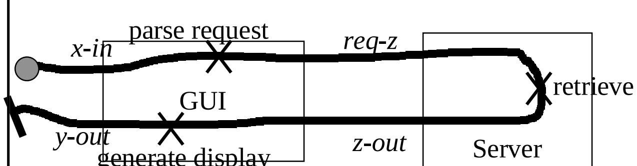
Complex UCM example

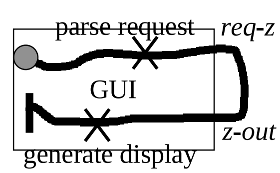
- group communi-cations system



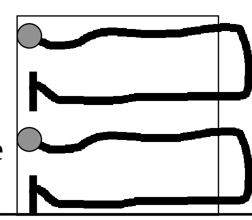
Component behaviour specification

- UCM breaks into fragments or sub-maps or factors, within a single component



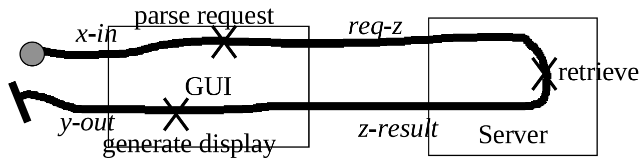


  - may be multiple Use Case Map fragments

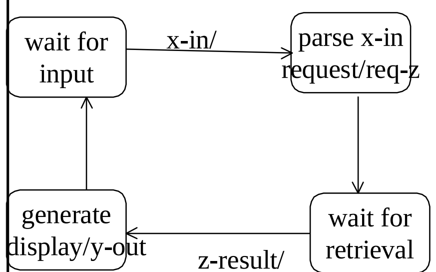


- module behaviour is specified by a state machine
  - UML StateChart, or SDL state machine

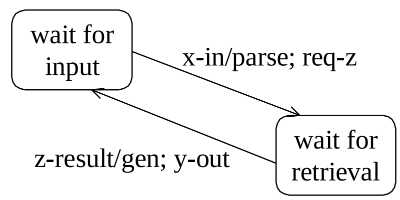
From UCM to module behaviour



States of the GUI module



A more condensed view, suppressing the activities parse and generate



Concurrency

- a major feature of real-time systems
- essential idea: ***concurrent activities*** are separately triggered, and possibly overlapped in time
- physical concurrency: simultaneous operation by separate processors
  - requires they be in separate tasks/processes
- logical concurrency: separate task/process or thread, permitting separate initiation and overlapping lifetime, even on the same processor
  - physical concurrency on separate processor, or separate member of a multiprocessor
  - threads are contained in the same process and can be physically concurrent on a multiprocessor
- need for concurrency: to react to separate events; to handle responses with overlapping lifetimes; to run in different places; to run in parallel on different processors

The path of development (Gomaa)

- Use case development
- static modeling of the domain of the application
- design objects of various types
- scenario analysis to state machines, for behaviour
- architecture as clustering of objects into subsystems
  – revised static model for data
  – and as concurrency

- *in this course*, we consider behaviour and architecture together
  – with enhanced scenario analysis via UCMs
  – capability to experiment with, or analyse, **concurrency issues**

Going over the design steps in UML
(1) Use Cases

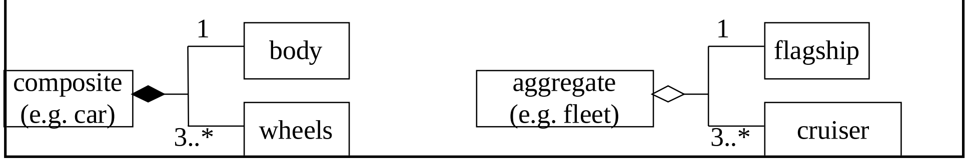
Gomaa’s bank example....

(1) ... use cases: fig 19.1... specification in words sec 7.6 or 19.2
  – actors (users or external entities (sensor, I/O device))
  – preconditions
  – description in words as a numbered sequence of steps
  – alternatives
  – postconditions
- use cases describe external requirements
- a use case may ***include*** another, as in fig 19.1, Validate PIN

Steps... (2) static modeling

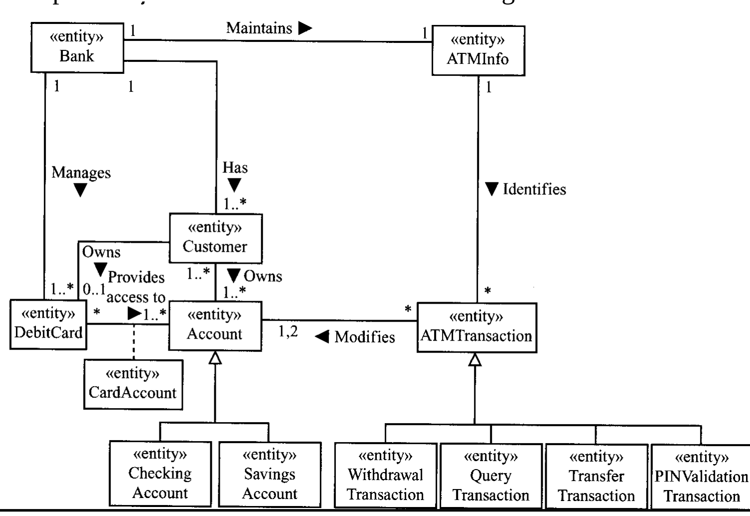
(2)... static modeling of the domain by entities, modeled as classes

- classes for types of information entities
- properties as data with types
- associations:
  - cardinality sec 8.1 p 140
  - ass’n description and direction and attributes
- composition: an instance of X consists of certain numbers of Y and Z
- aggregation: a composition where numbers may change
- inheritance



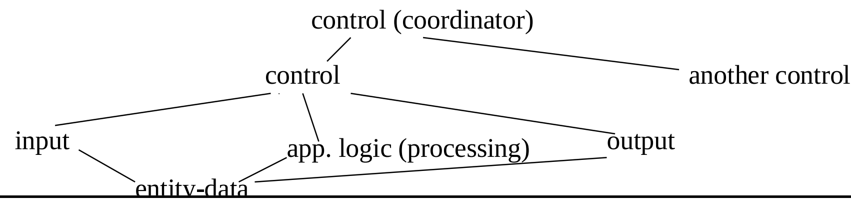
Steps.... static modeling of context

- entities representing system components
- relationships to information entities... as in Gomaa fig 8.18...



Steps... (3) design objects

- Gomaa introduces types of design objects fig 9.1
  - ***interface***
    - user, system, and device (in, out, in.out)
  - ***entity***, for data containers (and their operations)
  - ***control***, for program control logic
    - timer, state dependent control, coordinator for hierarchy
  - ***application logic***, for pure computation
    - business logic (less formal), algorithm (more formal)



Steps... design objects, from outside in

- fig 9.8, interface classes (bank card reader, receipt printer, user I/F, cash dispenser, operator I/F)
- entity classes (Account,
- control (ATM control of interfaces, fig 9.12)
- application logic (transaction mgrs for withdrawals, queries, transfers, PIN validation)
- coordinator of transactions
- packaging into subsystems

Steps... (4) scenarios to state-machines

- create a collaboration (or several) for each use case (fig 11.3)
  - design objects, numbered messages with arguments, conditions
  - numbering scheme: 1st for the UC, point, sequence number, letter for alternatives.
- optionally make a sequence diagram (e.g. fig 11.5)
- in state-dependent control objects, consider the flow of messages and decisions and create a state machine to enforce the logic of the flow (fig 11.6).
- separate analysis of separate cases (fig 11.7, 8, 9)

  ....combined (fig 11.10)
- hierarchical state machine for abort (fig 11.11)

but: HOW DO WE KNOW WE HAVE A GOOD SET OF CONTROLLER OBJECTS? (we will use UCMs for this analysis)

Steps.... (5) architecture

- this can be understood in different ways
  - in Gomaa's book, it is limited to clustering objects into subsystems, so it comes after the "flat" design of the objects.
    - subsystem level collaboration diagram fig 12.6 p 263
- Hofmeister/Soni/Nord describe an earlier and more prominent role:
  - ***conceptual architecture*** is the first pass, something like Gomaa. However, it is considered to need revision, before being a design.
  - ***module architecture*** is the second pass and fixes the modules and subsystems.
    - Control objects can be split and re-arranged, for instance.
  - ***execution arch*** is the third phase and assigns processes/threads and other concurrency factors,
  - ***architecture of code*** is the final phase (file structure for code, for instance)

Architecture in Gomaa's Bank System

- Clusters of objects, once they are identified

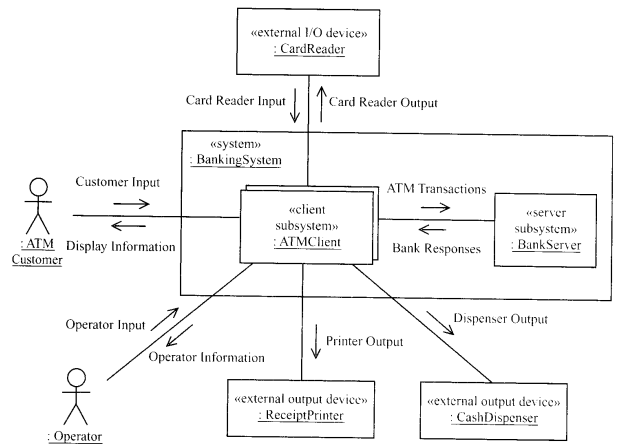


Figure 12.6 *Example of subsystem design: high-level collaboration diagram for Banking System*

State Machines for Behaviour Modeling

- a state machine is used (instead of code) to model the behaviour of one concurrent entity
  - Task in Gomaa or in many OS's; Process in UNIX
  - Active object in UML, or Capsule in RoseRT
- defines how the task reacts to input messages or signals, both as to computation and as to sending out messages and signals of its own.
- Task is a sequential program and has one state at a time
  - the state represents its present status and memory of the past
  - the task also may have data, which gives it an *extended state*
- State changes through transitions triggered by events (such as messages)

Basic state machines

transition trigger can be: ... input message event (of a given type)

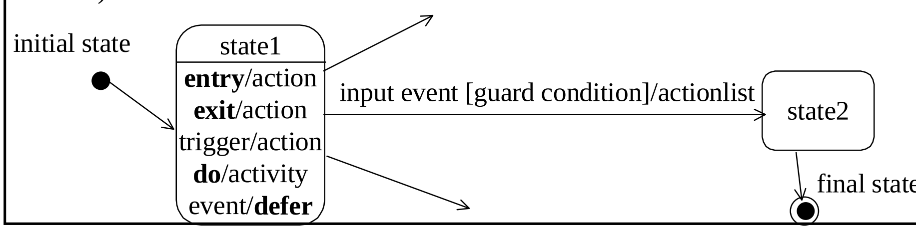
- input message type with guard condition on process data state and/or message data
- none (completion of a computation done while in the state)

action can be:... computation

- send a message type

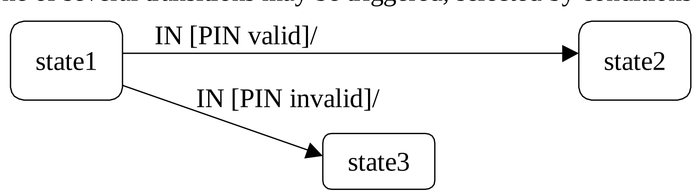
  (notice how a message type implies a port, which is a local view of the destination of a message)

state may have an entry computation, body (do) computation, exit computation, internal trigger/action pairs, and deferred events (that are held back while in this state)



Basic state machines (2)

- basic features are fairly conventional
  - an object (capsule in RRT) is always in one state,
  - a transition from a state may be triggered by receiving an event such as IN, or by completion of an activity executed in the state,
    - these possibilities can't be mixed
  - one of several transitions may be triggered, selected by conditions



  - in traversing the transition, a list of atomic actions may be executed...

examples

- look for...
  - actions on transitions, on states, on entry or exit
  - message sending actions (not obvious in first examples, as they are for free-standing controllers that do not communicate)
  - self-transitions
- think about
  - termination (why not shown in first examples)

State Machine example (Gomaa)

- numbered events show event, or event/action, keye to Use Case
- assumes initially i cruise mode when engine switched o

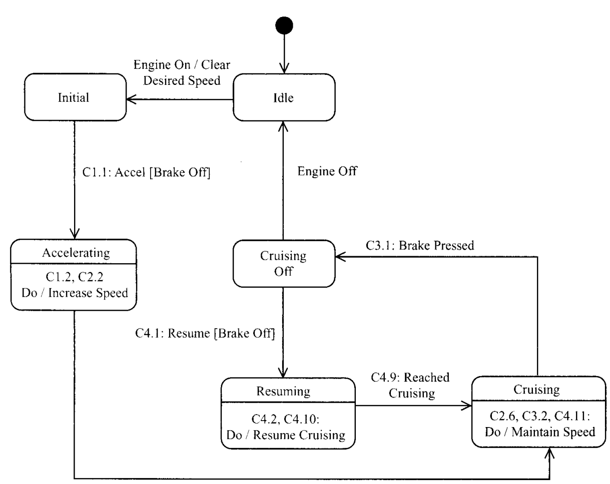


Figure 11.21 *Cruise Control statechart: Cruise Control scenario*

more detailed version...

- including hierarchical states for transitions out of automated control and engine running

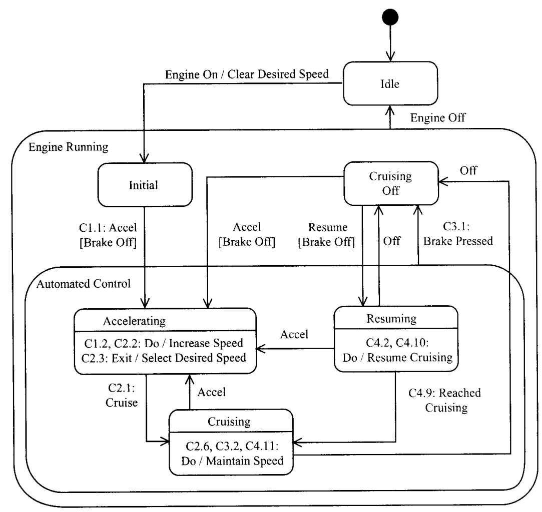


Figure 11.22 *Hierarchical Cruise Control statechart with all states and transitions*

more complex state machine (elevator)

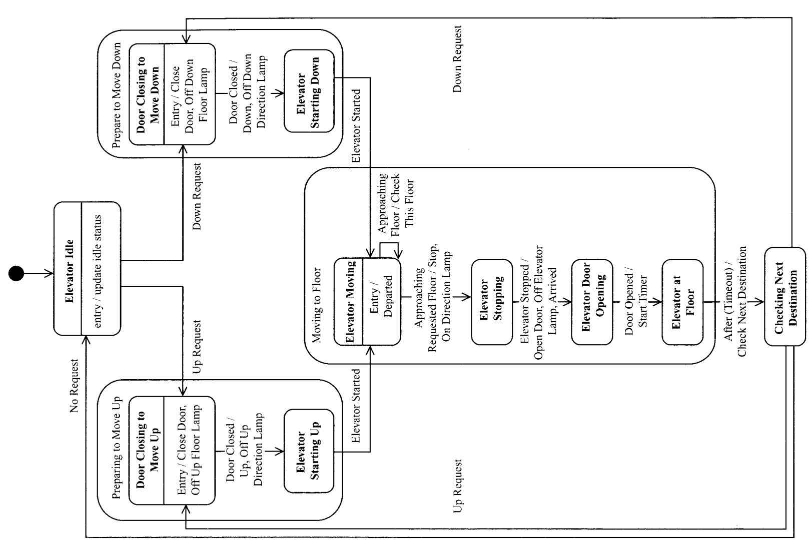
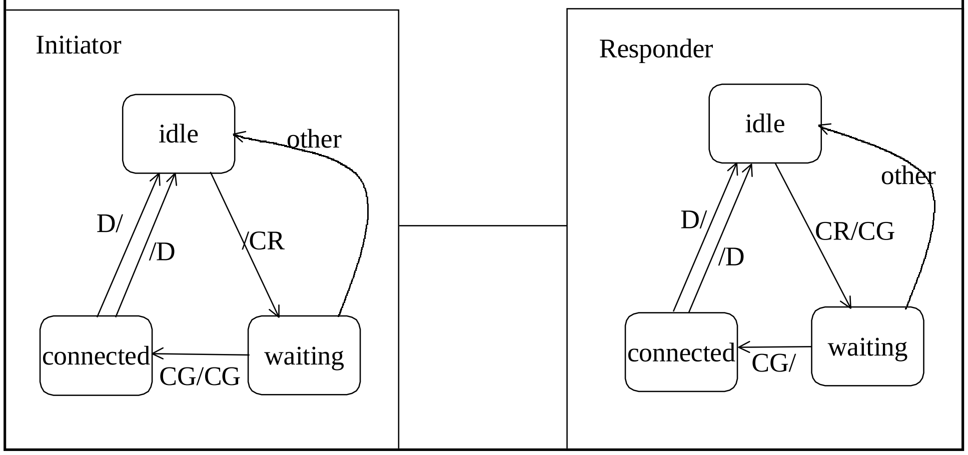


Figure 18.13 *Hierarchical statechart for Elevator Control*

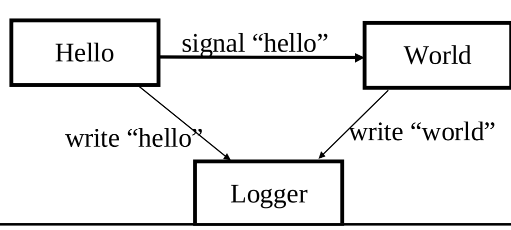
State machine example with two processes:
three-way handshake for connection establishment

- deals with request collisions; events include a timeout in waiting states, CR connection request, CG connection grant, D disconnect

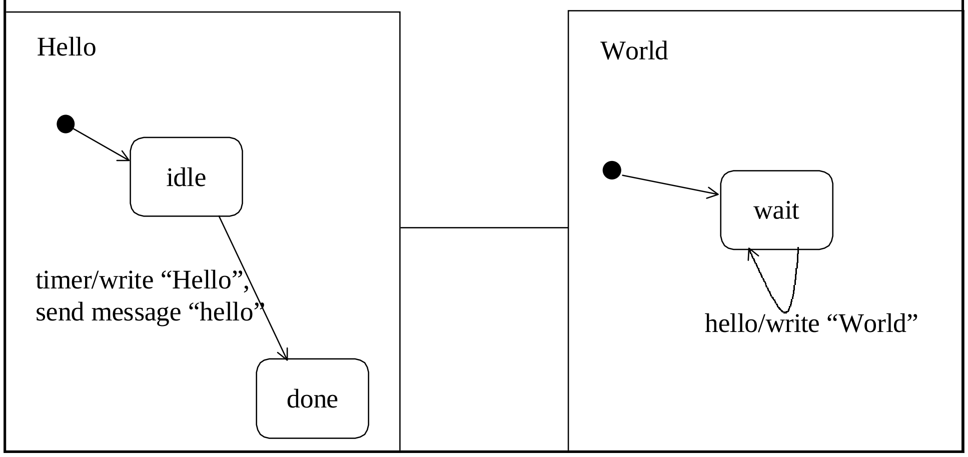


Learning to use ROSE-RT: first exercise

- exercise....
  - Build a program with two tasks, one of which prints out Hello, the other prints out World.
  - Focuses on active objects (Tasks) called *Capsules (*with state-based control, like Gomaa’s controllers), sending messages via *Ports*, using a set of signal types called a *Protocol*
  - Capsules have behaviour defined by *state machines*



Hello-world state machines

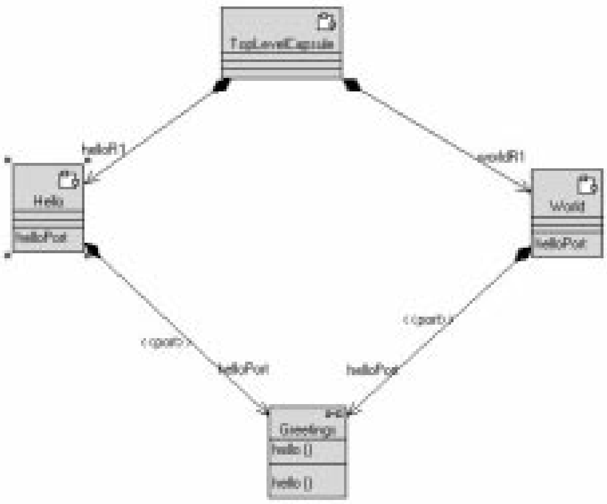


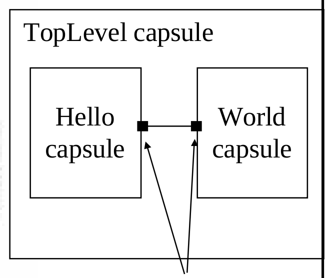
Hello world exercise

- create three capsules: TopLevel, Hello and World
  - compose Hello and World into TopLevel, which makes the capsules nested inside TopLevel
- create a protocol class called Greetings
  - with in and out signals typed as *hello*
  - compose the protocol into both Hello and World classes
    - each composition is stereotyped as a port, communicating via the protocol
    - the protocol defines the signals that can be handled
- open the structure diagram for TopLevel and connect the two ports together
- open the state diagram for capsule Hello and put in a state machine that
  - initializes to idle with a timer to go in five sec
  - on the timer, sends a hello signal to World, and writes the string “hello” to the log (using a canned protocol for logging).
- do the same for World, to receive the signal hello and write the string “world” to the log.

RRRT Display of Structure

UML Class view.................. structure diagram view





helloPort (two of them) employing the Greetings protocol with signal hello in and out

Build and run Hello-world

- build has a component wizard to create a component class with one instance, that will run in a simulated environment.
- run the single instance and see the output.

Heirarchical State machines

- great contribution of David Harel, inventor of StateCharts
- a state can have substates (p 19 Fig 2.7):
  - a transition to the state can go through into an inner state.
    - RRT uses a junction point on the boundary of the outer state to carry the transition from the outer to the inner diagram,
  - or, when the outer state is reached it can activate an Initial state in the inner machine
- the object is in both the outer state, and one of the inner states
  - some events are processed by the transitions in the inner machine, which describe detailed behaviour
  - such a transition can also leave the inner machine, to another state in the outer machine (normal completion)
  - an event that causes a transition away from the outer state destroys the inner state machine (abort)
    - (some models retain the inner state as a HISTORY feature, and on next entry go to it).
  - nesting can proceed to any level

Hierarchical state machines (StateCharts)

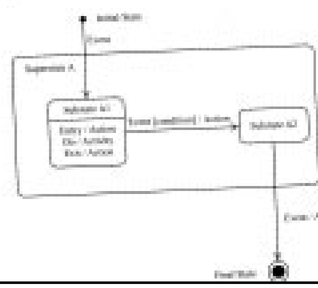
- adds the concept of substates:
  - *substates* in an outer state: state = (outer state).(inner state)
    - a transition *to* the outer state starts the inner state machine

      (a) in its initial state, or

      (b) in the state last occupied by the inner machine ("history" feature... the initial state is blank and labeled H)
    - a transition *away from* the outer state (a) aborts or (b) pre-empts the inner state machine
      - and a transition can go from an inner state directly to some other state outside... this also aborts the inner machine, but allows multiple next states to depend the inner state value.
  - *concurrent* inner state machines, all active while in the inner state (*AND* machines). Models parallel operation of part of an algorithm.

StateCharts example

- this state shows three sets of actions the object may carry out:
  - Entry/ list of atomic actions
  - Do/ list of activities that may take time and be pre-empted
  - Exit/ list of atomic actions
- capsule objects are created in a standard state “initial” and spontaneously, when started, execute a transition from there.
  - conditions could be used to choose from several initial transitions
- capsule objects have a standard Final State, and a transition to that state destroys the object.
- p19 Fig 2.7 in Gomaa...



Concurrent sub-states

- this is an oddball feature of Harel’s StateCharts that is not supported by RRT
- entry into a superstate can initiate multiple inner state machines, that are all active at once
  - used to describe algorithms with internal concurrency
  - equivalent to a UCM AND-Fork/Join
  - p20 Fig 2.8

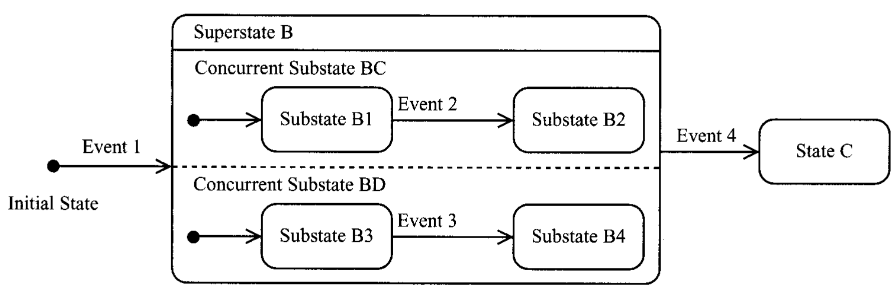


Figure 2.8 *UML notation for statechart: superstate with concurrent substates*

StateCharts example...

Thinking in state machines

- a state machine can imitate a flow chart
  - operations in states, decisions governing transitions
  - states to wait for input
    - send a prompt or request on entering the state, exit on arrival of input
  - decisions as conditional transitions

- a machine with a single state is possible
  - state waits for input
  - different transitions from the state, depending on the signal type
  - processing as an action on the transition, followed by sending output, and return to the state.
    - any kind of processing action can legally be defined on a transition, however the concept of action as atomic is more suitable for short actions than long ones that take time.
    - for instance, an action on a transition cannot be interrupted or aborted by arrival of a signal.

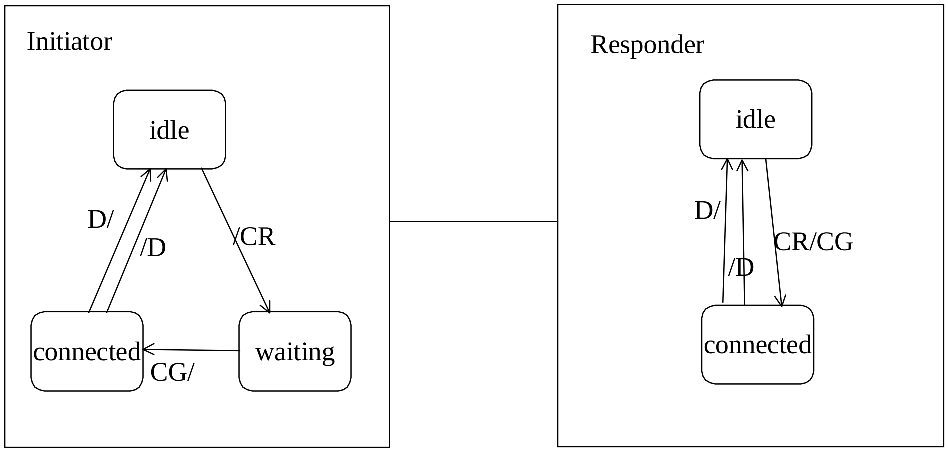
Value of state-machine specification

- Why state machines
  - ... instead of flow charts, for instance
- heavily used in communications software
  - protocols, call-processing
- introduced to support verification of logic of interacting processes
- the ***global state*** of a system is the vector (x, y, z...) of states of the individual processes (state x of process 1, state y of process 2,...).
- as events are sent from one process and received at another, they are handled by FIFO channels; global state includes the messages in transit.

Correctness analysis of state machines

- as messages are sent and received:
  - ... the global state changes can be constructed, this is called ***reachability analysis***.
    - State explosion limits the usefulness of this approach.
    - also, asynchronous message sending can lead to unbounded channel contents
- the global state machine can be analyzed to see if it
  - is guaranteed to reach a terminal state, or
  - has a deadlock.
  - well-known tool PROMELA does this, for instance
- state machines give some practical clarity to complex sequences
  - spaghetti code lives on as state machines.... intentional

State machine problems



- CR received while waiting after sending ones own CR (request collision)
  - ... fixed by sender timeout
- lost CG... inconsistent state (one idle, one connected)

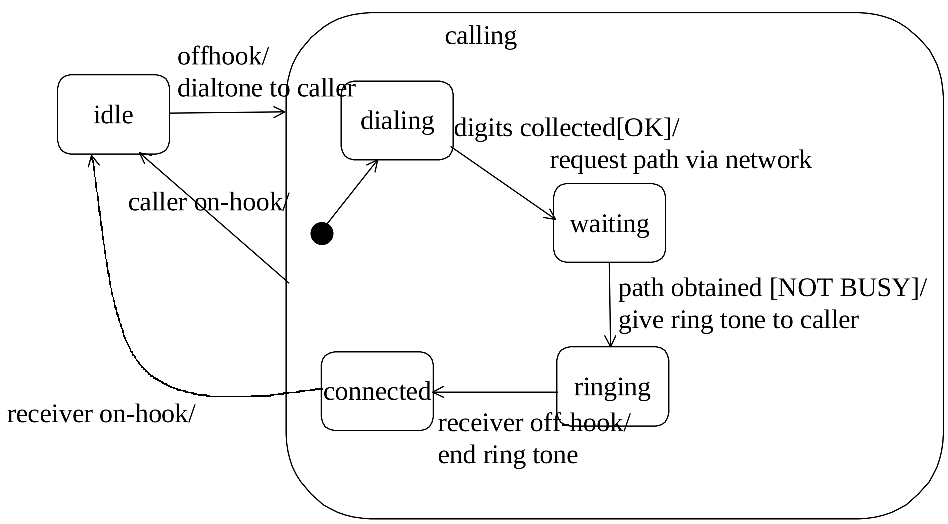
(L4-5) Call processing and state machines

- lecture overview...
- sm for a scenario entire
- sm for a piece... interacting sms
- concept of global state
- elevator control state machines

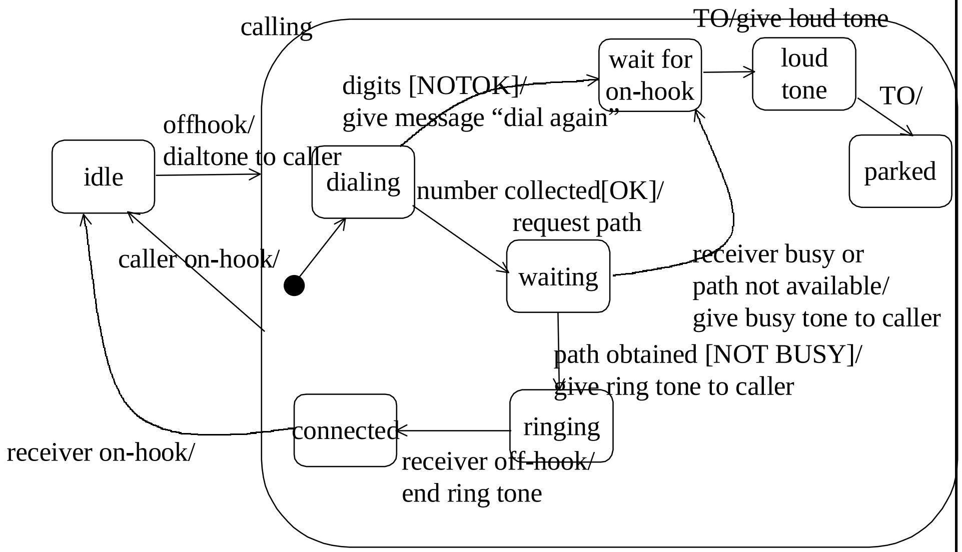
Basic Call processing

- Use case:
  - off-hook at originator, give dial tone
  - collect digits from originator
    - one-at-a-time, gives an opportunity for detailed states
    - or, consider as a single activity under local control of some kind
  - once all collected, check if receiving party is free
  - if receiver is free, find a route that is available and set ringing tone at originator and at receiver
  - if off hook at receiver, connect the voice path and stop ringing at both ends
  - if on-hook at either end, disconnect the call.
- alternatives:
  - if no route is available, give busy signal to originator and abort the call.
  - if receiving party is not free, set busy tone at originator
  - if originator goes on-hook, abort the call.

Basic call processing... state machine



Call processing... more conditions



Modeling

- call object, with originator, receiver, route
- originator terminal interface object, and terminal device
  - interface object sets dial/ring/busy tones and collects digits
- originator control object, receiver control object
  - messages on-hook, off-hook, dialed-number from terminal interface objects to control
  - messages set-dial-tone, set-ring-tone, set-busy-tone, and reset etc, to interface objects
  - messages connect-req, on-hook from orig to recvr
  - busy, not-busy, off-hook, on-hook from recvr to orig
  - message find-route from orig to network, with return messages route-found(route) or no-route
- receiver terminal interface object and terminal device

Call processing sequence

- fig XXXX

94.586 Object-Oriented Design of Real-time Systems...Winter 2001 A-62
Copyright C. M. Woodside 2000

Call processing state-machines

- orig, recvr XXXX

second RRT exercise (L4)

- simplified call processing model (call model #1)
  - hardwired terminals, one to originate, one to receive
  - off-hook, ring, off-hook, talk, on-hook, buzz remaining terminal, on-hook
  - model the terminal interfaces and the control processes.
- inject events for terminal off-hook and on-hook, through a test port
  - inject the events through a probe at each end

(L6) Some typical behaviour definitions

- SMs to go with pipelines, client-server, single independent tasks.
- simple cases
- Asst 1.

(L7-8-9) Developing an architecture

- outline of the problem
- important role of scenarios

UCMs for scenarios

- semantics of the UCM model
- reasoning about UCMs
- Navigator tool

Developing the architecture

- using UCMs
- Hofmeister styles and issues
- criteria for concurrency and for modularity
- Craig example

(L10-11-12) Hierarchical systems

- more complex cases from L6
- multiple controller systems
  - controller and workers (elevator)
- client-server (e-commerce
- callback interactions
- getting there from UCMs with multiple fragments in a task
- router example
- elevator

(L12-13) Pipelines

- signal processing?

Call processing pipeline (DFC)

(L14) Peer interaction systems (e-commerce)

- async interaction

(L15) Layered arch

- layer as a service layer, abstraction

(L16-17) Control systems

Issues