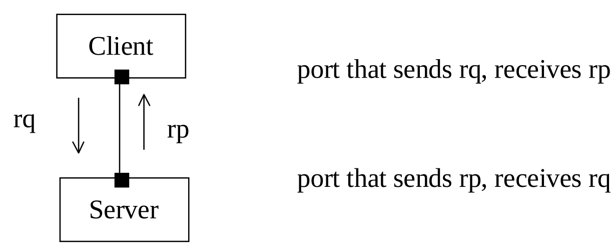
Object-oriented Design of Real-time Systems

State Machines for Behaviour Specification

- **Examples**
  - **–behaviour of a single process or a small set of processes**
  - **–behaviour of common process roles**
    - **server, tiered server, forwarder**
    - **pipeline**
  - **–continuous-time controller**
  - **–call processing**
  - **–connection protocols**
- **Understanding**
  - **–diagnostics, heuristics, abstraction**
  - **–formal system analysis by reachability analysis**
- **Expression in SDL**

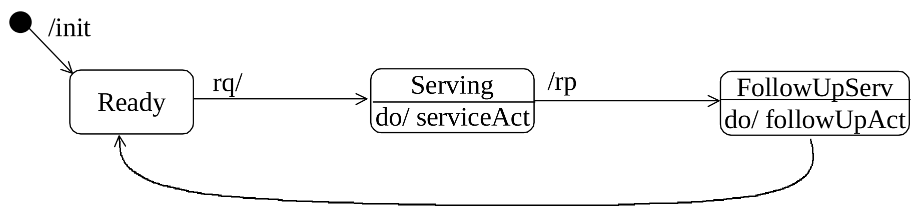
Behaviour of common process roles: Server

- Architecture



- Use case:
  - Server receives a request event of type *rq* with arguments *x*
  - Server executes activity *serviceAct* as local execution within the Server process (active object), with result *y*
  - Server sends reply of type *rp* to requester, with data *y*
  - Server executes activity *followUpAct*

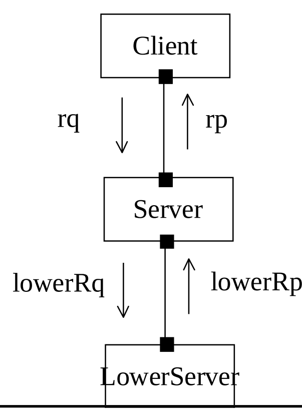
Server SM



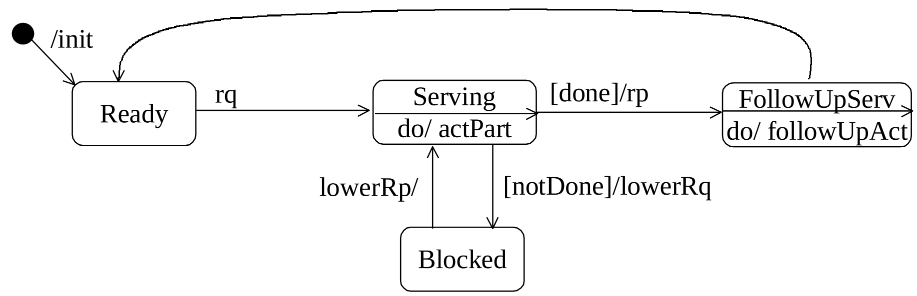
- rq can include data to bring in arguments, rp can include data for results
- multiple service types... the type can be part of the argument x
- exceptions are passed back in reply data y
- init activity on creation of the server
- terminate command not included here (may be OS operation, not graceful)... graceful programmed terminate could act in ready state

Tiered server

- During the service activity, Server makes a request to some LowerServer
  - there may be several LowerServers, and multiple requests
  - requests may be conditional

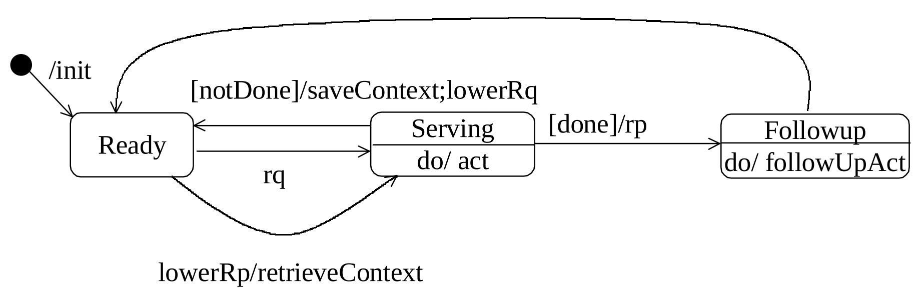


Tiered server SM



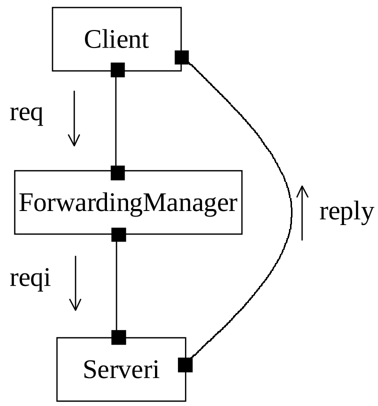
- messages rq and rp could still have parameters not shown here
- guard condition “done” refers to act
- ? how to show that act pauses and resumes?... possibly several times?
- followUpAct may also contain nested requests

Async tiered server (virtual threads)



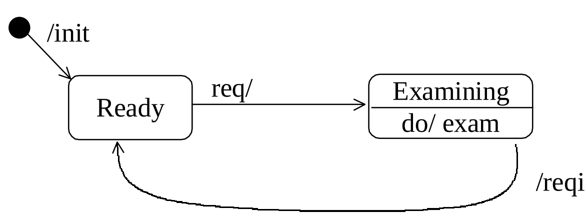
- when a lower service must be requested, the context of the current operation is saved in a data table. Thus the server is able to handle additional requests instead of blocking.
  - this is logically equivalent to a multi-threaded server
- followUpAct may also contain nested requests

Forwarding Manager



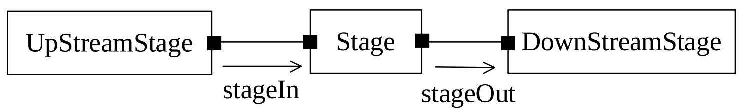
- Use case:
  - Client sends request to FM
  - FM examines the request, decides how it should be handled, and sends it to a chosen Server (FM may partially process it also)
  - Serveri executes the requested operation, sends reply to Client
  - Serveri finishes any followup activities

Forwarding manager SM...

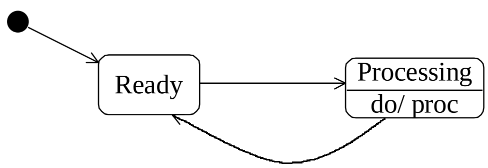


- notation assumes that the out-port for reqi is uniquely defined to go to the chosen Serveri
  - to be complete, the alternative message types would have to be explicitly defined, with separate transitions... the modifier i is symbolic.
- reqi passes the original Client identity along to Serveri, to provide a return address.

Pipeline Stage



- Stage receives a message of type stageIn, processes it and sends the result as a message of type stageOut



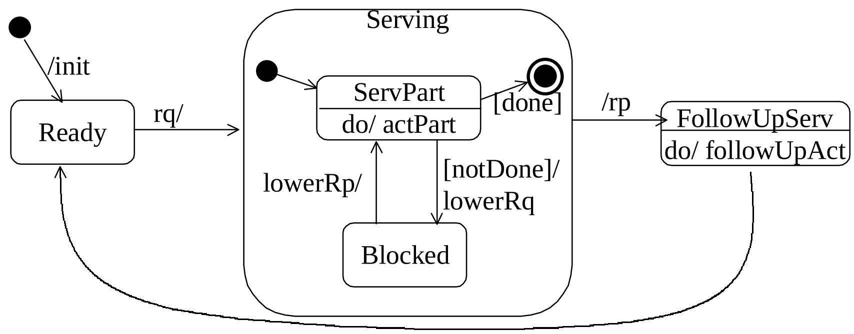
Questions not addressed

- termination
- exceptions that must be reported elsewhere
- aborts

- still, the service paradigm is relatively simple...

Behaviour abstraction to get the Tiered Server from the Server

- the details of the main part of the service can be nested inside the Serving state, and inside followUpServ



Service types

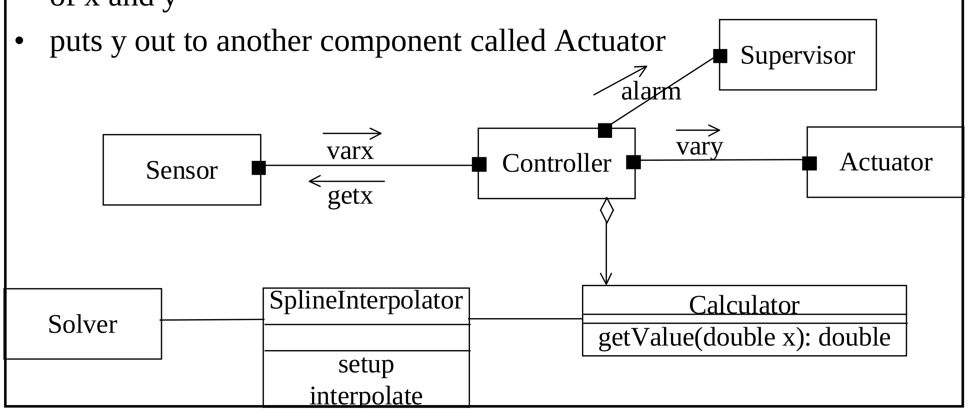
- it may be necessary to use different states to handle types of services
- common part can be covered by first state
- guard on the type gives transition to separate states by type for completion
  - at top level
  - or, into nested set of states with common handling of termination
- separate states by type can contain detail by type, or have additional transitions by type, including exception handling by type.

Inheritance

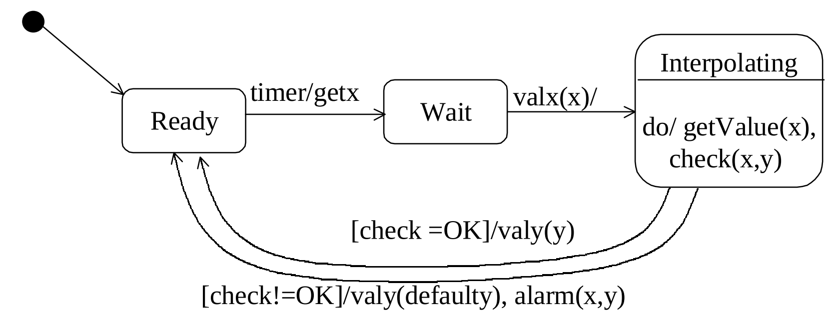
- tiered server state machine can be obtained by inheritance from Server
  – by adding nested states inside Serving
  – or by adding additional states and transitions at the same level

Controllers in Continuous Time: Interpolator

- takes in a variable x from Sensor
- determines a variable y from splined interpolation in a table of values of x and y
- puts y out to another component called Actuator



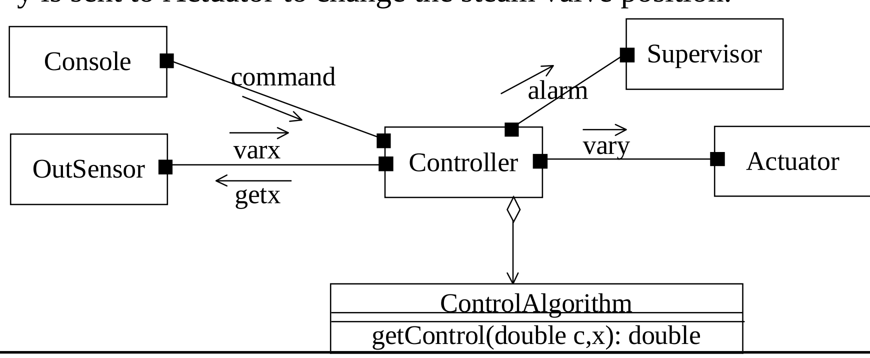
Interpolator SM



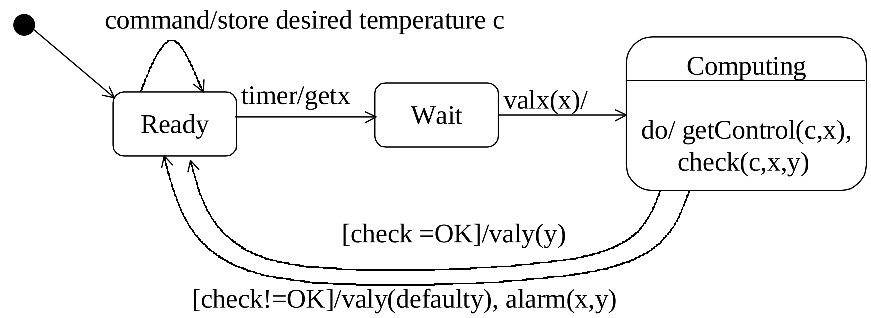
- interpolator may fail due to bad data or a bad sample from the Sensor; check tests both x and y for acceptability
- defaulty is a “safe” value of y which can be used by the actuator
- Supervisor handles the alarm by logging it and perhaps informing an operator (e.g. if there is a sequence of alarms)

Feedback Controller

- Actuator controls say a valve on a steam pipe; OutSensor is a temperature sensor; Console sends a desired temperature value, which is constant until changed again.
- On a timer, the temperature x is measured and compared to the desired temperature in command, and a new value for y is computed by ControlAlgorithm
- y is sent to Actuator to change the steam valve position.



Feedback controller SM



- notice alternative transition to catch commands
- notice sources of delay in reacting
- how is c set to begin with?... maybe a command to BEGIN, with initial value of c, and timer not set until then?
- command to vary timer might be appropriate... sent here?... command to suspend control?
- same approach shown for checking... again, faulty instruments may lead to haywire control calculation

State machine discussion

- event types
  - signal event... an asynchronous event from outside
  - call event... a syncrhonous event from outside
  - change event... a notification of a change in an attribute value, in or out
  - time event, either timeout or absolute.
- action (on transition, exit, entry) vs activity (do:)
  - an action is regarded as indivisible; it will not be left partly undone. some might view it as taking zero time, but that is not the essence (zero *logical* time, yes)
    - Douglass says the action may be by another object (a call)
  - if there is a list of actions, the list is indivisible
  - an activity executed on a state takes time and may be interrupted, or even left unfinished (e.g. by a higher level transition)
- an internal transition (or *reaction*) appears to have some ambiguity as when it will be executed: by interrupting the activity of the state, or between activities if there is a list?

State machines...

- pseudostates are non-state objects on the state diagram
  - initial state (black circle)
  - history state (circle with H) (as initial, but a transition INTO history resumes at previous state
  - deep history takes you as deep in as you last were
  - condition node: immediately examines guard conditions for next transition (no entry, exit, activity etc)
  - diagram connector allows to paste pages together
  - synchronization pseudostate is a counter that counts transition executions and enables or disables them depending on the counts
- inheritance
  - child SM can add events, states and transitions (not delete them),
  - retarget transitions
  - change actions and activities,
  - introduce nested states (refinements) but not change the nesting of an existing state

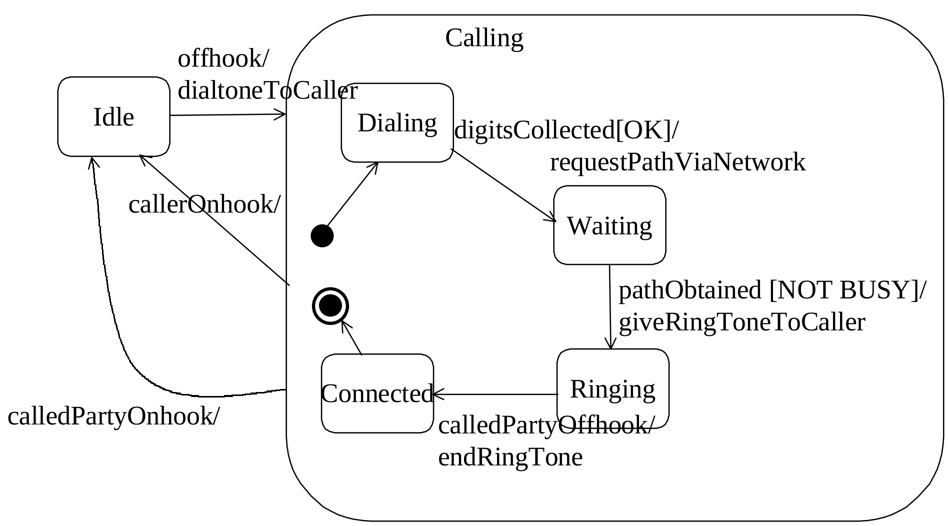
Ill-formed state models

- (Douglass p 330)
- lacking an initial state (needed at top level, and in nested SM if transition is to the outer state)
- guard on initial transitions
- conflicting transitions (same event for different transitions at same or different levels, guard that is ambiguous)
- guards on completion transitions (may never complete)
- guard is evaluated before the transition is taken (action on guard variables comes after)

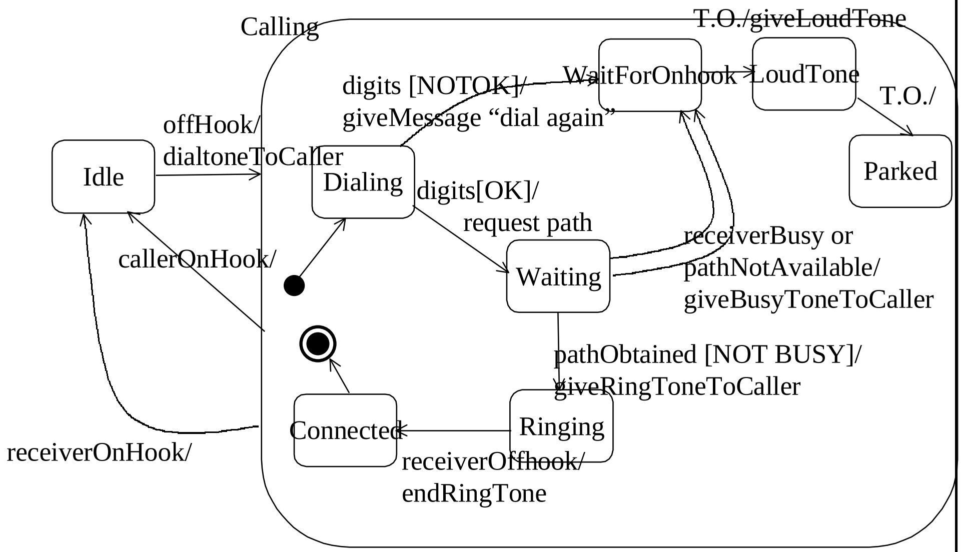
Basic Call processing

- 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 behaviour



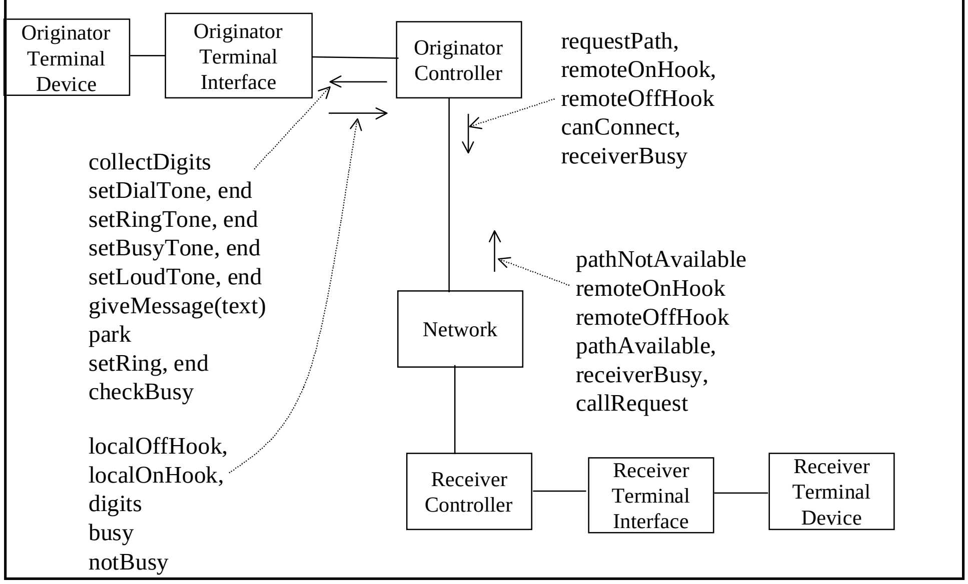
Call processing... more conditions



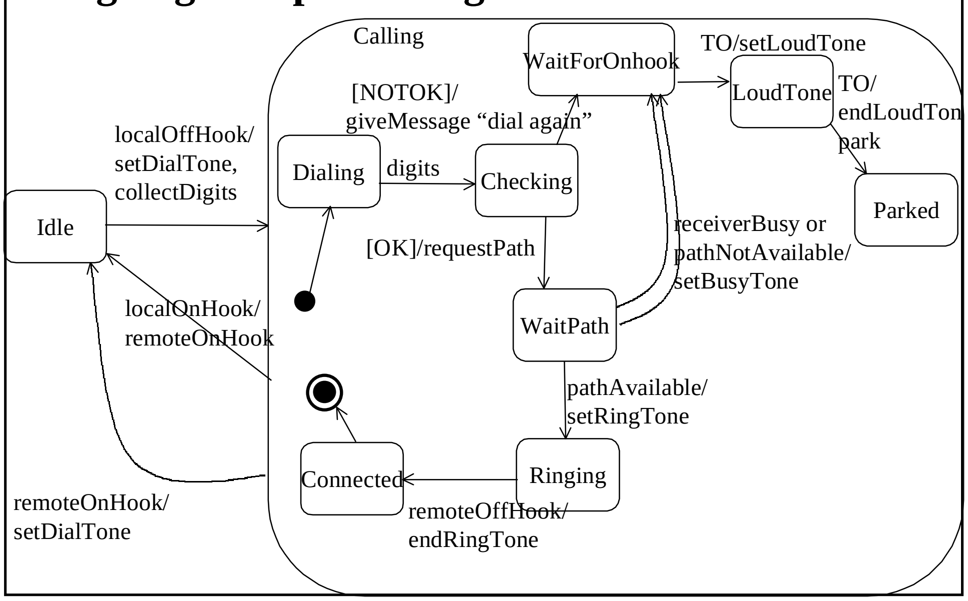
Structure for call processing

- originator, receiver, route objects
- 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

CP processes (minimal model)



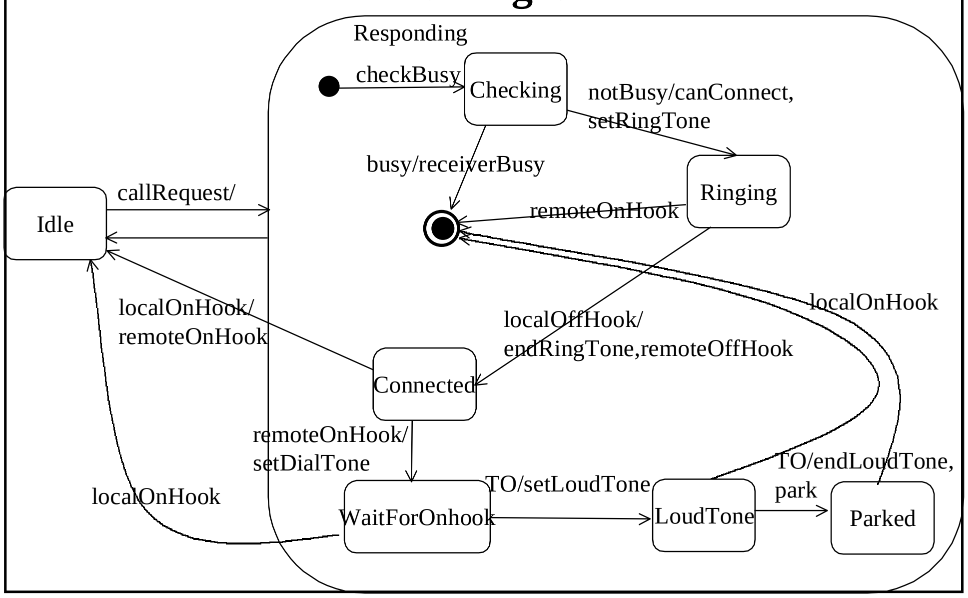
Outgoing Call processing SM bound to Controller



Comments

- now we have bindings of message events to connections, which implies ports and protocols

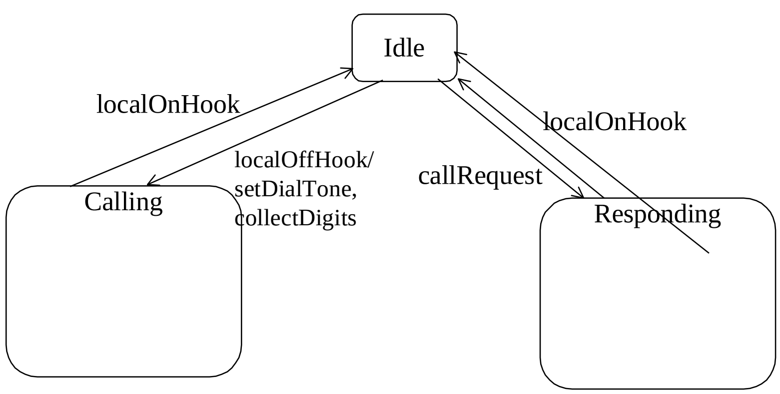
Incoming CP



Incoming CP discussion

- remote on-hook handling...
  - abort if before local off hook
  - after that, local set must still be dealt with if not put on-hook
- local on-hook handling... must be signalled if connection still exists
- ...should handle a race in on-hook

Incoming and Outgoing together



Exception Handling

Adding features

- Call forwarding by receiver
  - unconditional
  - on busy
  - on no-answer-after-N-rings
- Call waiting
  - signal receiver
  - wait until receiver on-hook, then ring
  - form a queue of calls?

Connection protocols

- one message:
  - originator sends CR (connection request) and goes to Connected state
  - acceptor receives and goes to Connected state
  - problems: one side crashes, or the message is lost
  - connection collision turns out OK except for confusion of who is originator (this affects the start of the connection operation)
- two messages:
  - add a CG (connection grant) from acceptor, and timeout-retransmit at originator
  - problems: originator crash leaves acceptor hanging, duplicate CR arriving after termination.
- three-way handshake:
  - add a final AK, and timeout-retransmit of CG at acceptor
- best to have a connection sequence number, to catch delayed duplicates of CR or CG
- TCP has a sequence number for both ends, ack'd both ways

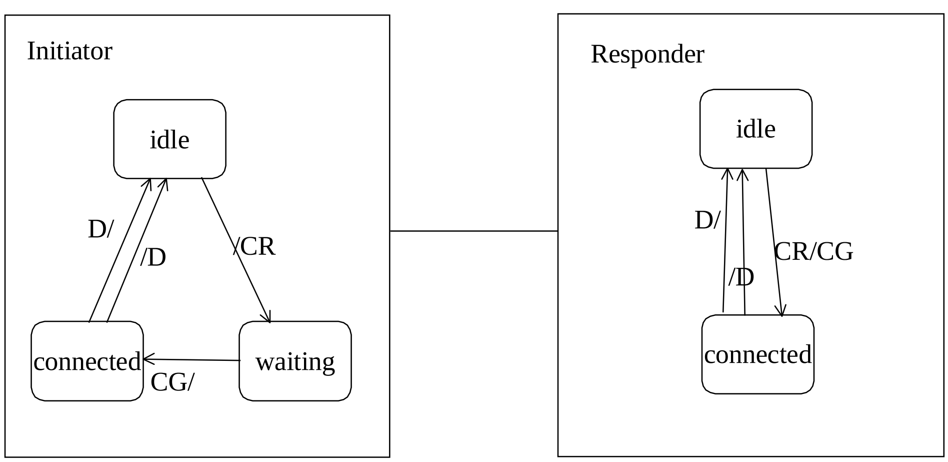
Value of state-machine specification

- 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 on each channel.

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 global state, or
  - has a deadlock, a state with no transitions out of it.
  - has a critical race, in which a trivial ordering difference in events produces markedly different behaviour
  - is sensitive to a message loss
  - 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)