ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

Network Protocol Design and Evaluation

04 - Protocol Specification, Part I

Stefan Rührup

University of Freiburg
Computer Networks and Telematics

Summer 2009

CoNe Freiburg

IIF INSTITUT FÜR INFORMATIK FREIBURG

Overview

- **In the last chapter:**
  - The development process (overview)

- **In this chapter:**
  - Specification
  - State machines and modeling languages
  - UML state charts and sequence diagrams
  - SDL and MSC (Part II)

What are we modeling?

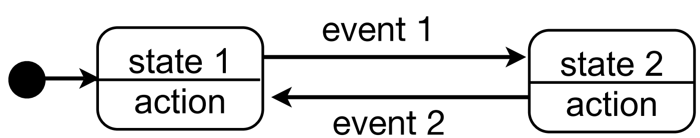
Transitional Systems	**Reactive Systems**
input-output transformation	event-driven
e.g. scientific computation, compilers	e.g. **communication protocols**, operating systems, control systems
correctness criteria: - termination - correctness of input-output transformation	correctness criteria: - non-termination under normal conditions - correctness of event-response actions

formal models describe event-response *sequences*, including state information

[S. Leue, Design of Reactive Systems, Lecture Notes, 2002]

Specification with State Machines

- **A protocol interacts with the environment**
  - triggered by **events**
  - responds by performing **actions**
  - behaviour depends on the history of past events, i.e. the **state**



- state machines do not model the data flow, but the **flow of control**

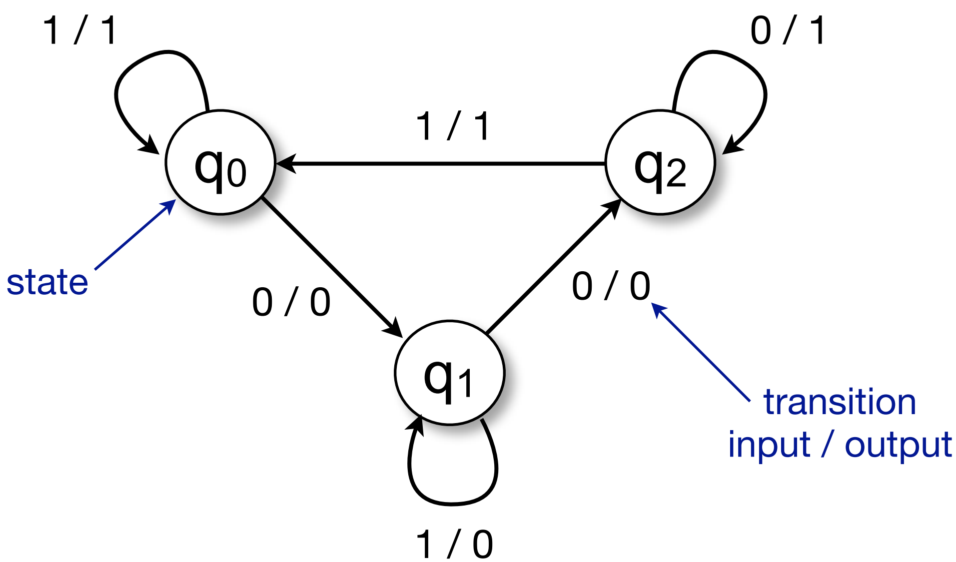
Specification with State Machines

- **Why state machines?**
  ...and not programming languages?
  - lack of formal semantics
  - risk of overspecification
  - requirements specification should be kept implementation-independent

Finite State Machines

- **Acceptors (sequence detectors)**
  - produce a binary output (yes/no) on an input sequence
  - accept regular languages
- **Transducers**
  - Mealy machines (output determined on current state and input)
  - Moore machines (output determined on current state)
  - both models are equivalent

Mealy Machines, Example



state diagram

State	In	Out	Next state
q_0	0	0	q_1
q_0	1	1	q_0
q_1	0	0	q_2
q_1	1	0	q_1
q_2	0	1	q_2
q_2	1	1	q_0

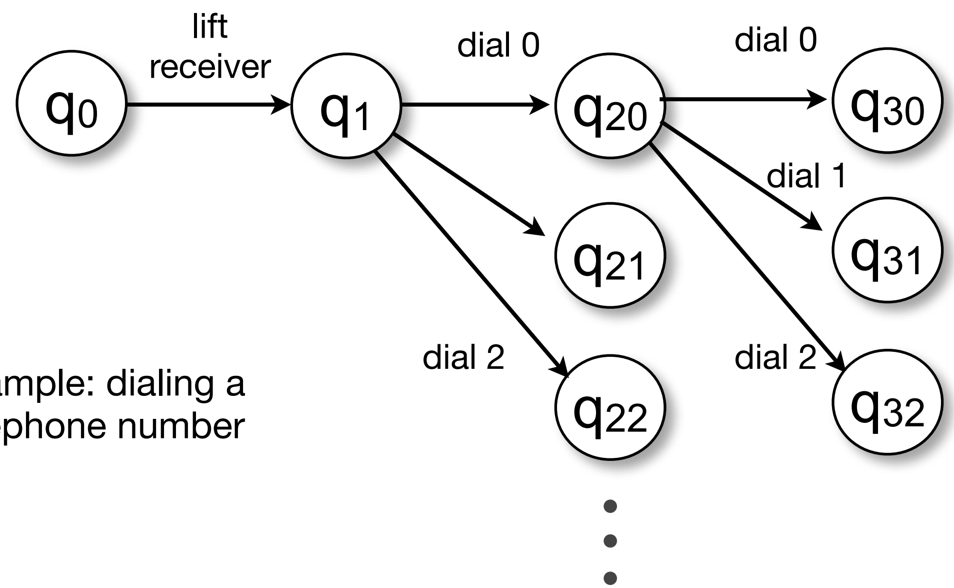
state transition table

Definition of a Mealy machine

- A *Mealy state machine* is a tuple (Q,q_0,I,O,T,G), where
  - Q is a finite, non-empty set of states,
  - $q_0 \in Q$ is the initial state,
  - I is a finite set called the input alphabet,
  - O is a finite set called the output alphabet,
  - T is a transition function, $T: S \times I \rightarrow S$, and
  - G is an output function, $G: S \times I \rightarrow O$.

Limitations of FSMs

- **No data variables**
  variable values and changes have to be coded into the state space
  → exponential state space explosion



Example: dialing a telephone number

Limitations of FSMs

- **Problem with finite memory:**
  - finite variable range
  - problem when modeling communication channels:
    - size of the channel unknown
    - determining buffer size = overspecification

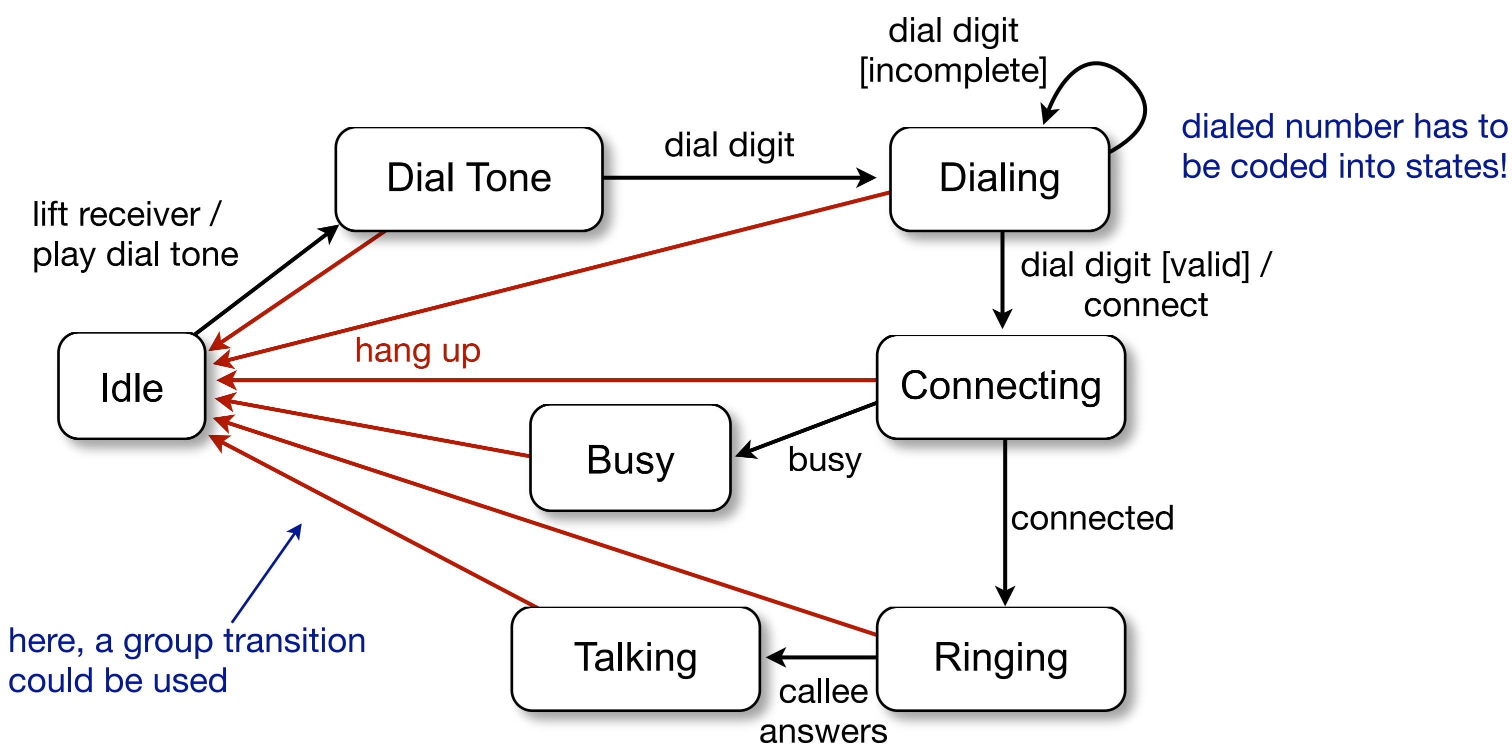
Limitations of FSMs

- **Problem with concurrent FSMs**
  - no communication channels
  - no synchronization
  - composition of interacting FSMs leads to new states and an explosion of the state space

- Communication protocols can be seen as concurrent state machines

Limitations of FSMs

- **Missing abstraction, missing composition**



State machines for specification

- Original FSMs are not suitable for modeling and specifying processes in distributed systems
- Extended state machine models:
  - Communicating Finite State Machines
  - Harel statecharts (superstates, concurrent states)
  - Extended Finite State Machines (variables, operations, conditions)
  - Basis for many practical modeling and specification languages such as SDL, UML.

Description Languages: Structure vs. Behaviour

- **Structural languages:**
  - describe the static, structural concept (architecture)
  - e.g. class diagrams, component diagrams
- **Behavioural languages:**
  - describe behaviour, i.e. activities, interaction
  - e.g. state machines and sequence diagrams

Description Languages: Constructive vs. reflective

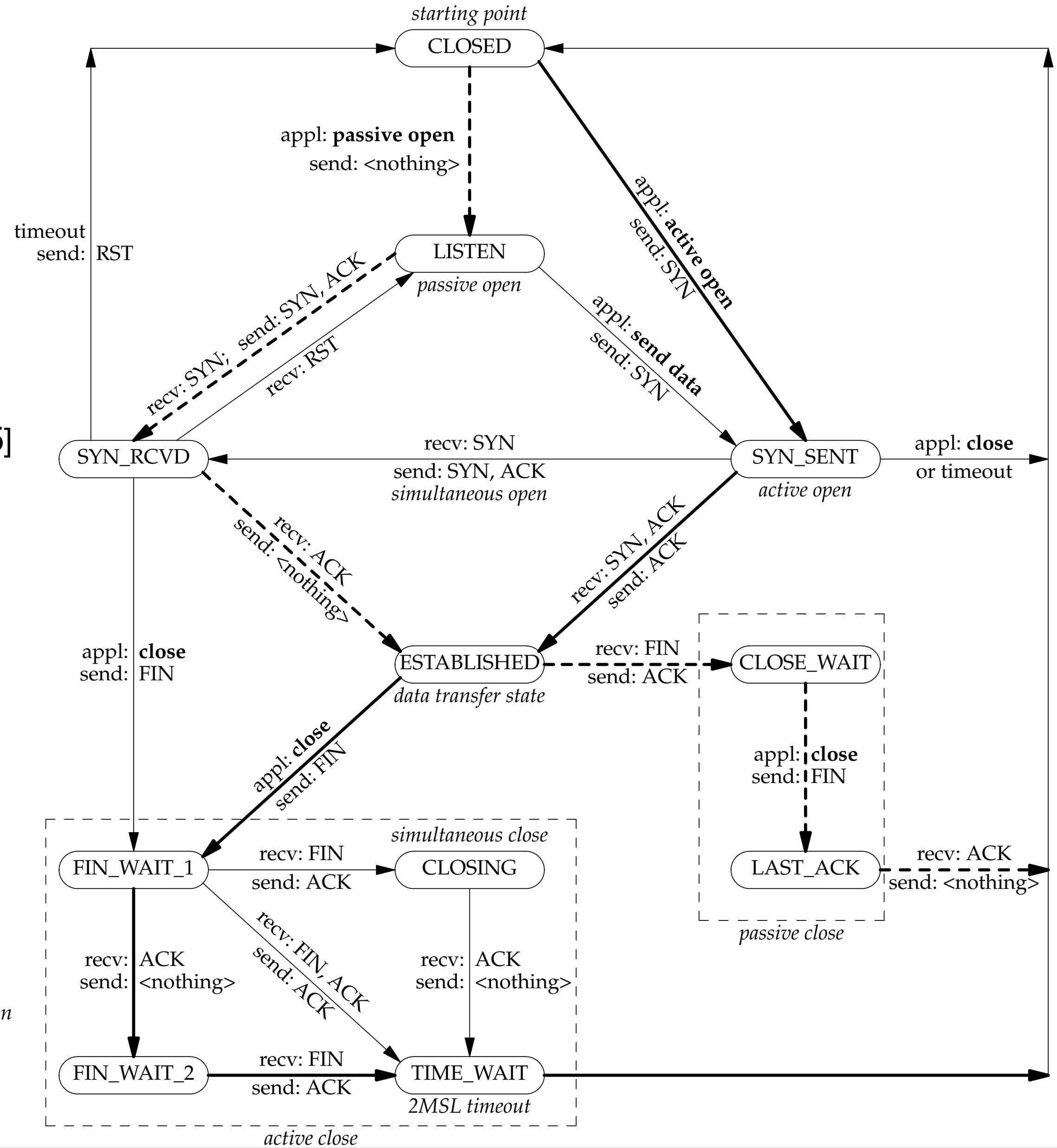
- **Constructive languages:**
  - describe information for executing the model or for (executable) code generation
  - e.g. state machines

- **Reflective or assertive languages:**
  - describe views of the model, statically or during execution
  - e.g. sequence diagrams

[D. Harel: “Some thoughts on statecharts, 13 years later”, 1996]

Example:
TCP state transition diagram

[Wright, Stevens: “TCP/IP Illustrated, Volume 2: The Implementation”, 1995]



normal transitions for client
normal transitions for server
appl: *state transitions taken when application issues operation*
recv: *state transitions taken when segment received*
send: *what is sent for this transition*

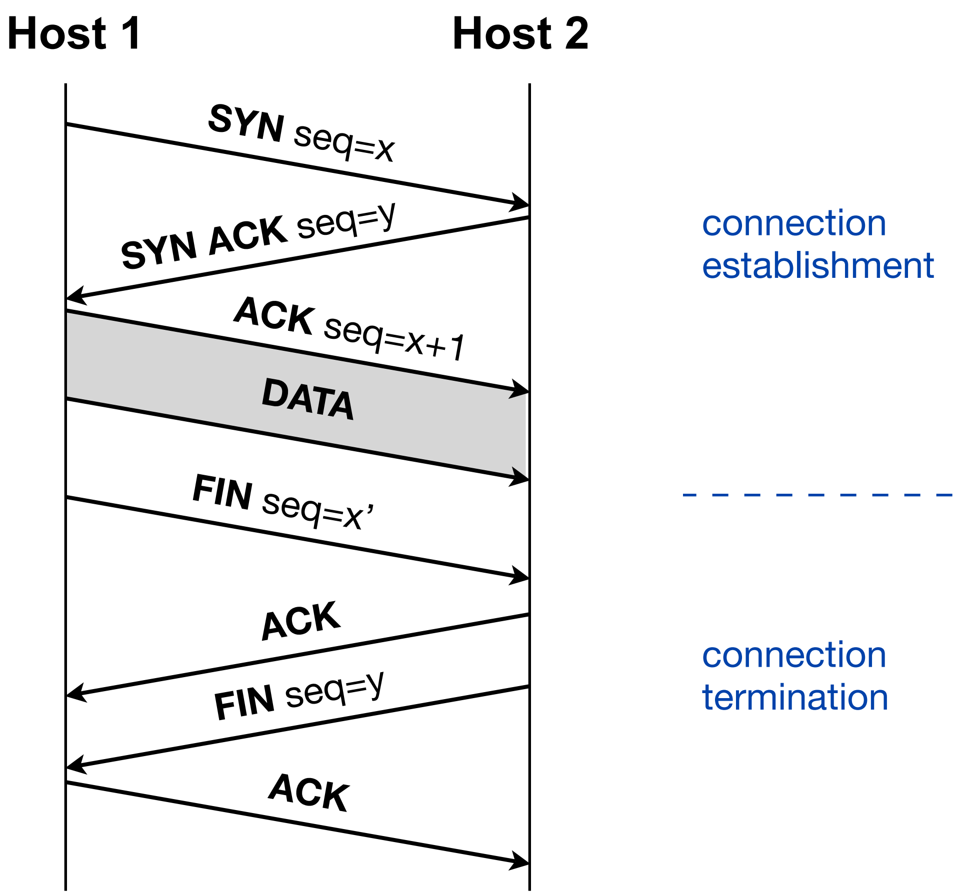
Example:

TCP connection state diagram

[RFC 793]

```
                              +---------+ ---------\      active OPEN
                              |  CLOSED |            \    -----------
                              +---------+<---------\   \   create TCB
                                |     ^              \   \  snd SYN
                   passive OPEN |     |   CLOSE        \   \
                   ------------ |     | ----------       \   \
                    create TCB  |     | delete TCB         \   \
                                V     |                      \   \
                              +---------+            CLOSE    |    \
                              |  LISTEN |          ---------- |     |
                              +---------+          delete TCB |     |
                   rcv SYN      |     |     SEND              |     |
                  -----------   |     |    -------            |     V
 +---------+      snd SYN,ACK  /       \   snd SYN          +---------+
 |         |<-----------------           ------------------>|         |
 |   SYN   |                    rcv SYN                     |   SYN   |
 |   RCVD  |<-----------------------------------------------|   SENT  |
 |         |                    snd ACK                     |         |
 |         |------------------           -------------------|         |
 +---------+   rcv ACK of SYN  \       /  rcv SYN,ACK       +---------+
   |           --------------   |     |   -----------
   |                  x         |     |     snd ACK
   |                            V     V
   |  CLOSE                   +---------+
   | -------                  |  ESTAB  |
   | snd FIN                  +---------+
   |                   CLOSE    |     |    rcv FIN
   V                  -------   |     |    -------
 +---------+          snd FIN  /       \   snd ACK          +---------+
 |  FIN    |<-----------------           ------------------>|  CLOSE  |
 | WAIT-1  |------------------                              |   WAIT  |
 +---------+          rcv FIN  \                            +---------+
   | rcv ACK of FIN   -------   |                            CLOSE  |
   | --------------   snd ACK   |                           ------- |
   V        x                   V                           snd FIN V
 +---------+                  +---------+                   +---------+
 |FINWAIT-2|                  | CLOSING |                   | LAST-ACK|
 +---------+                  +---------+                   +---------+
   |                rcv ACK of FIN |                 rcv ACK of FIN |
   |  rcv FIN       -------------- |    Timeout=2MSL -------------- |
   |  -------              x       V    ------------        x       V
    \ snd ACK                 +---------+delete TCB         +---------+
     ------------------------>|TIME WAIT|------------------>| CLOSED  |
                              +---------+                   +---------+
```

Example: TCP Sequence Diagram



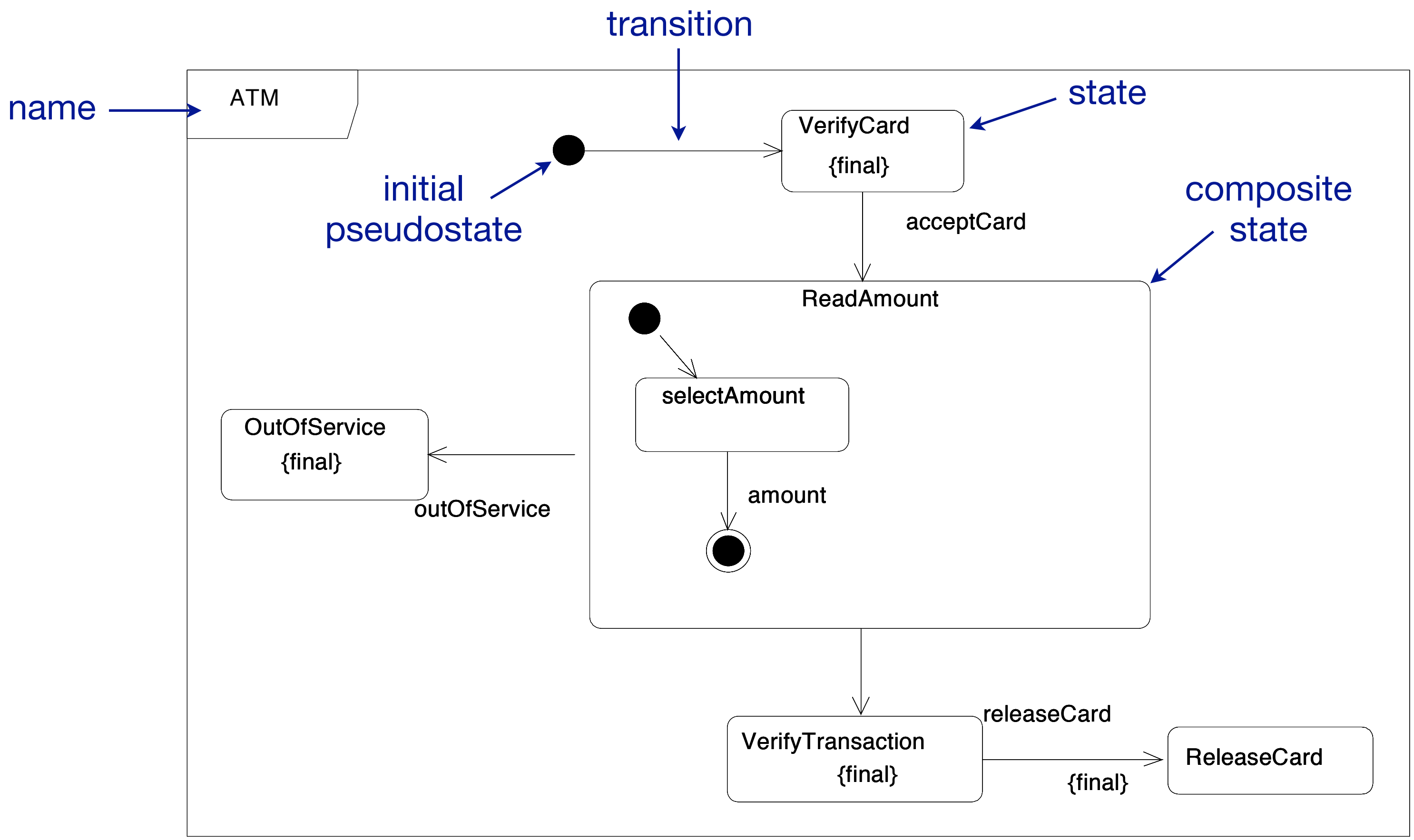
UML

- **Unified Modeling Language**
  - general-purpose language for modeling and specification in software engineering
  - in this context of particular interest: State machines, sequence diagrams

- **The standard**: UML 2.0 Superstructure Specification
  http://www.omg.org/spec/UML/2.0/

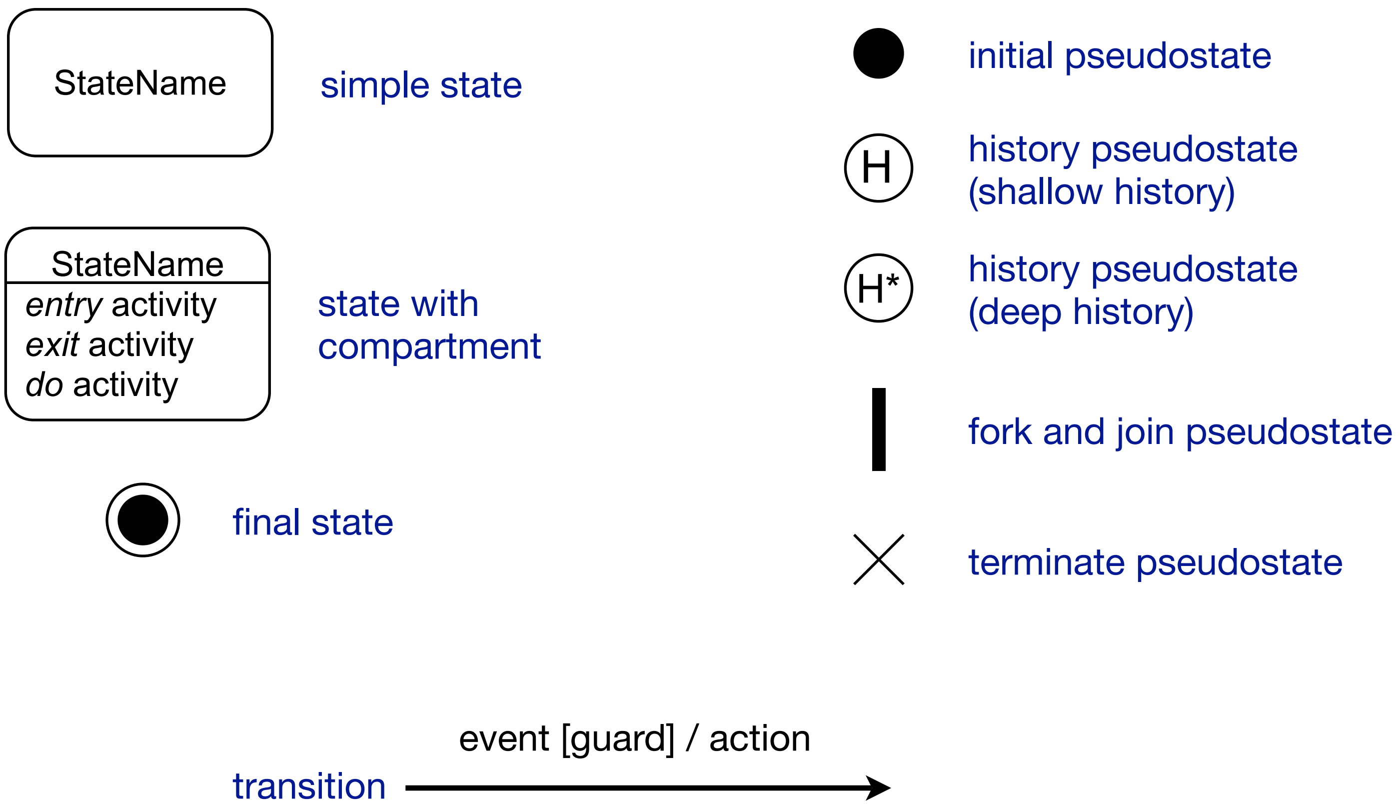
- see also: Lecture on *Software Design, Modelling and Analysis in UML* by Bernd Westphal, Uni Freiburg

UML State Machine, Example

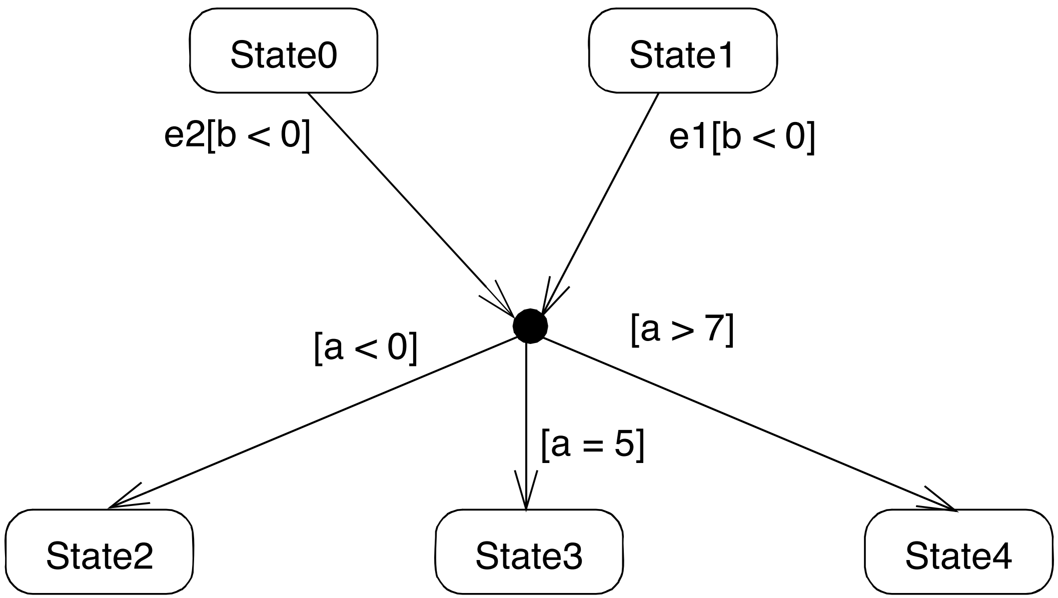


[UML Superstructure Specification v2.2]

States and Transitions



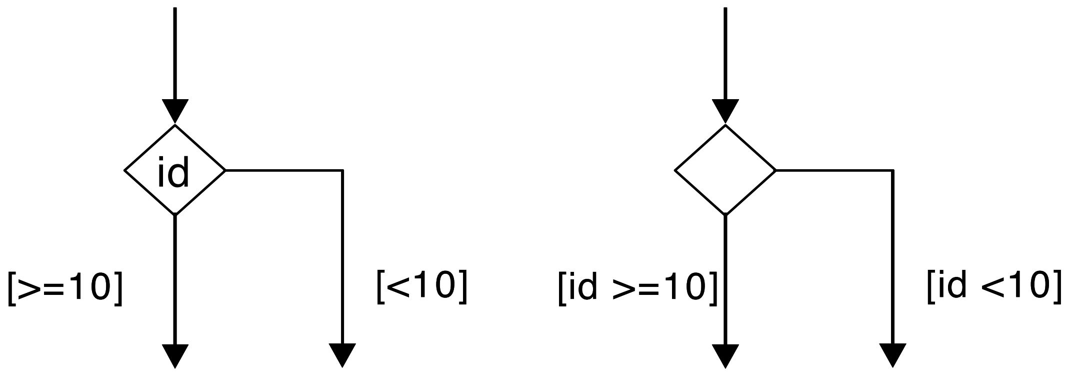
Junction pseudostates



junctions realize *merges* or *static conditional branches*

[UML Superstructure Specification v2.2]

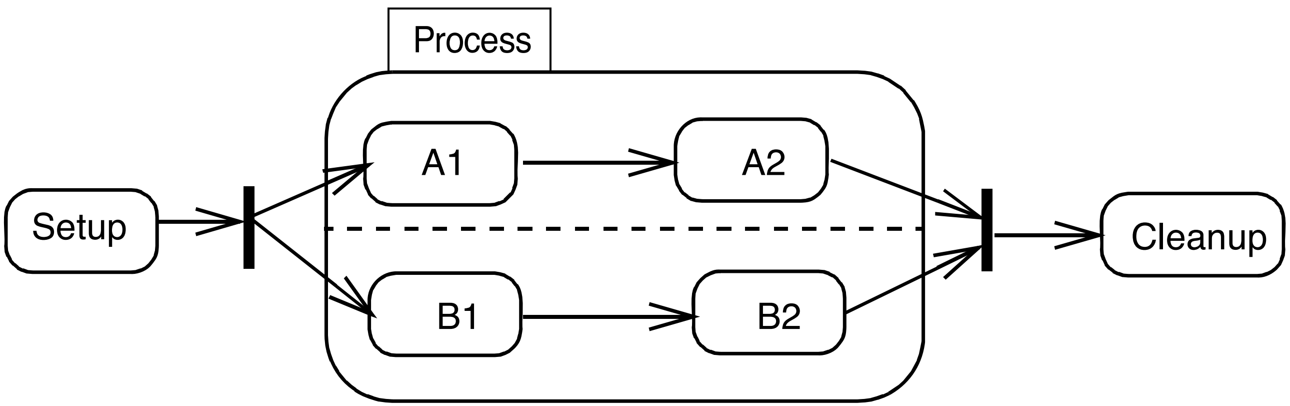
Choice pseudostates



choices realize *dynamic conditional branches*

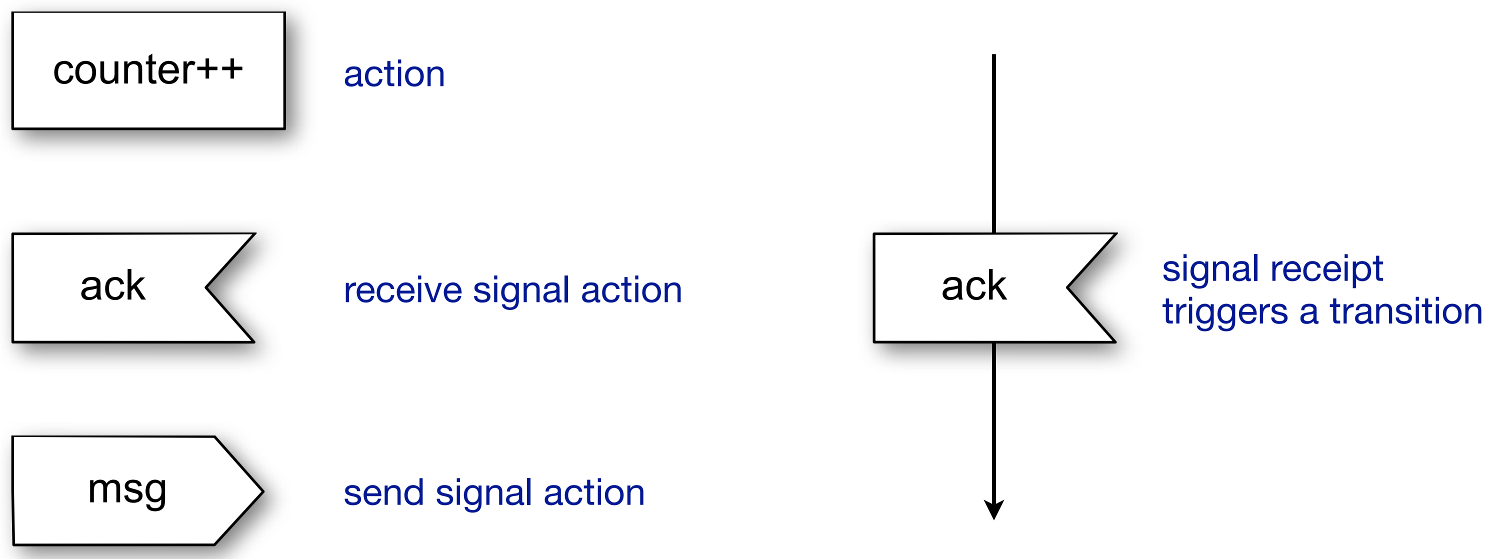
[UML Superstructure Specification v2.2]

Fork and Join

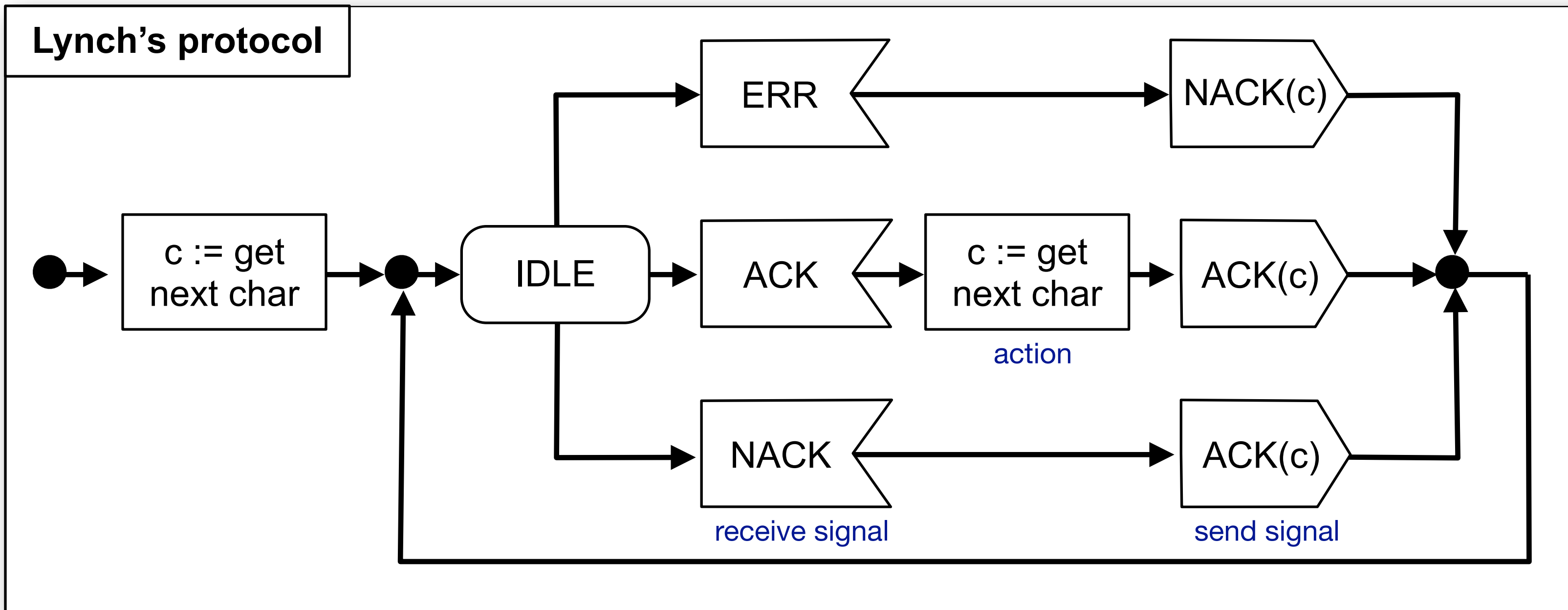


[UML Superstructure Specification v2.2]

Actions


counter++
action
ack
receive signal action
msg
send signal action
ack
signal receipt
triggers a transition

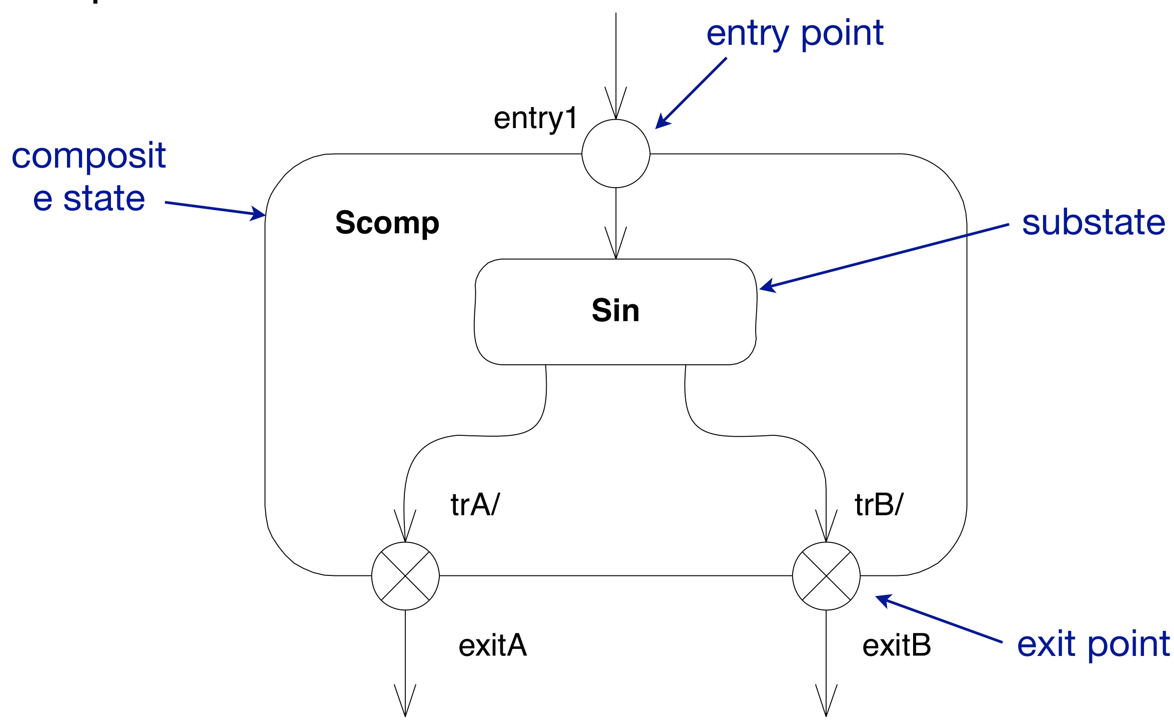
Example



Lynch's protocol
c := get next char
IDLE
ERR
NACK(c)
ACK
c := get next char
ACK(c)
action
NACK
ACK(c)
receive signal
send signal

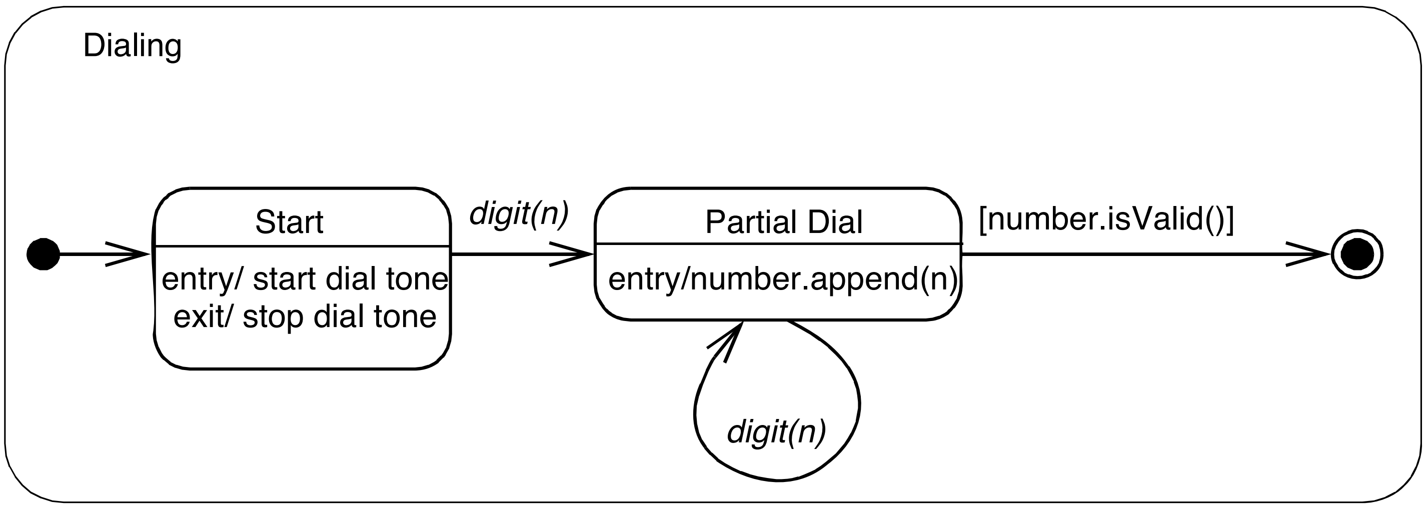
Composite states

Example:



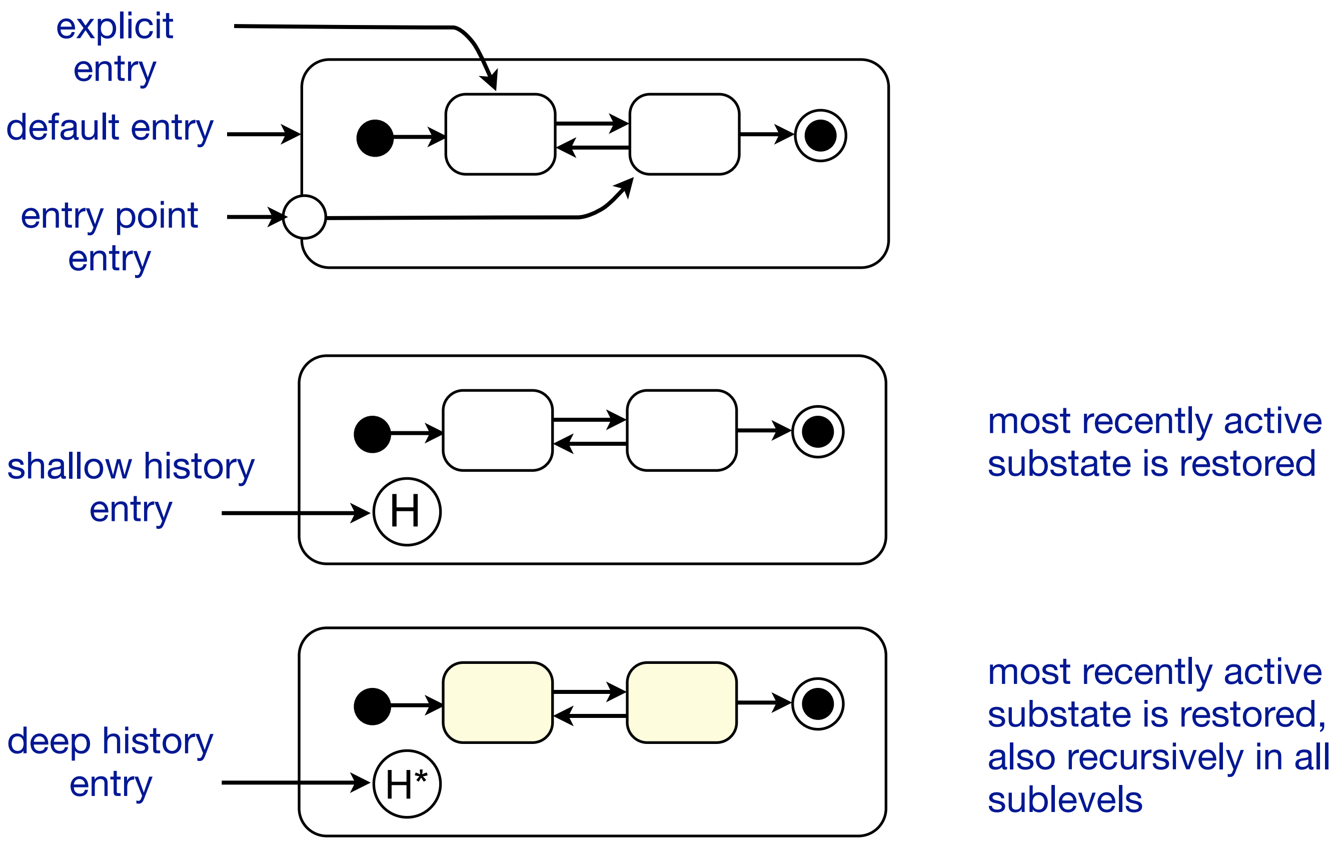
[UML Superstructure Specification v2.2]

Composite states, Example

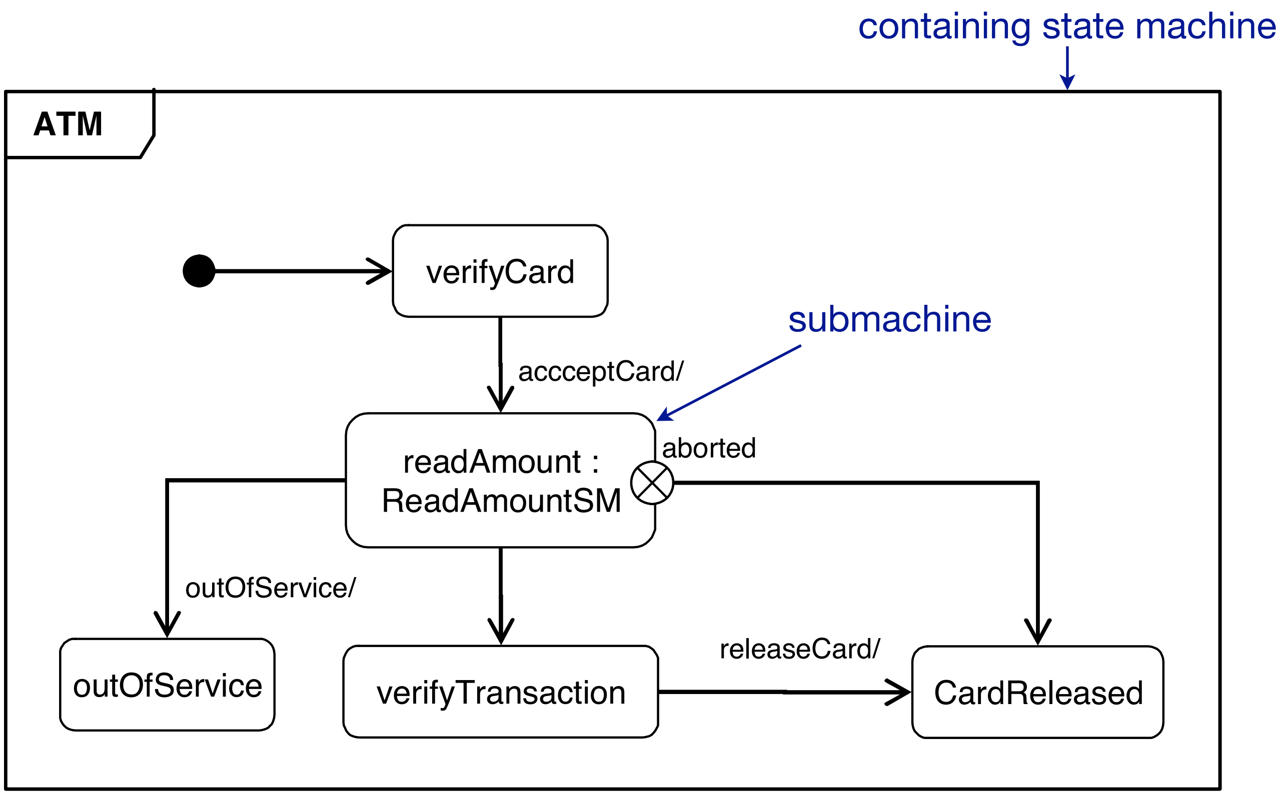


[UML Superstructure Specification v2.2]

Substate entry



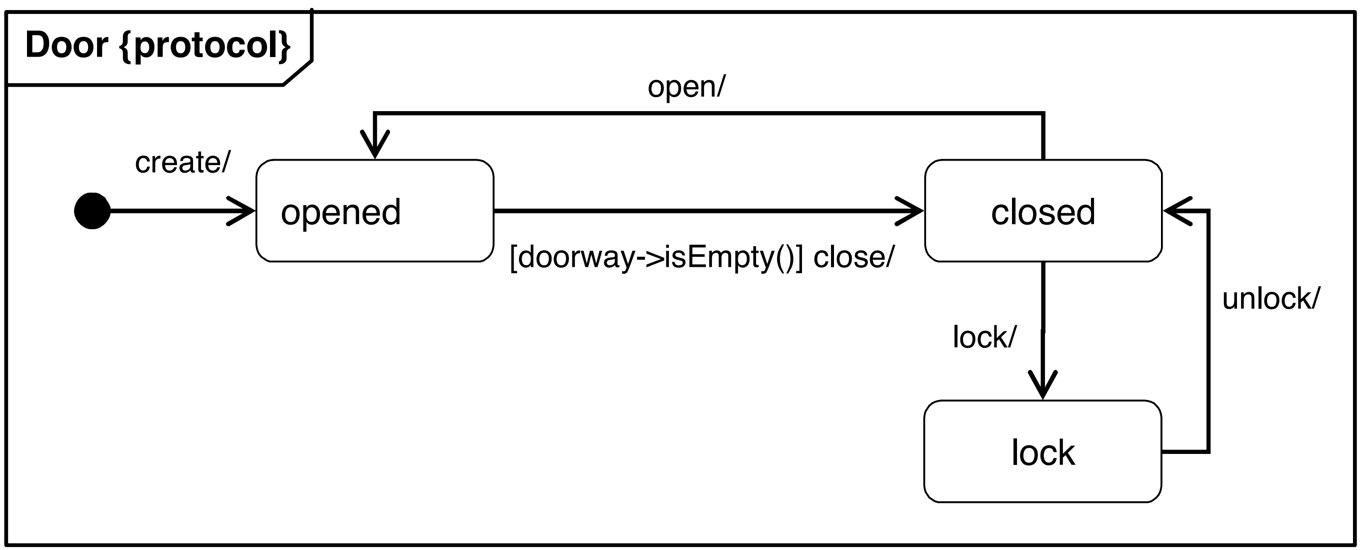
Submachine states



[UML Superstructure Specification v2.2]

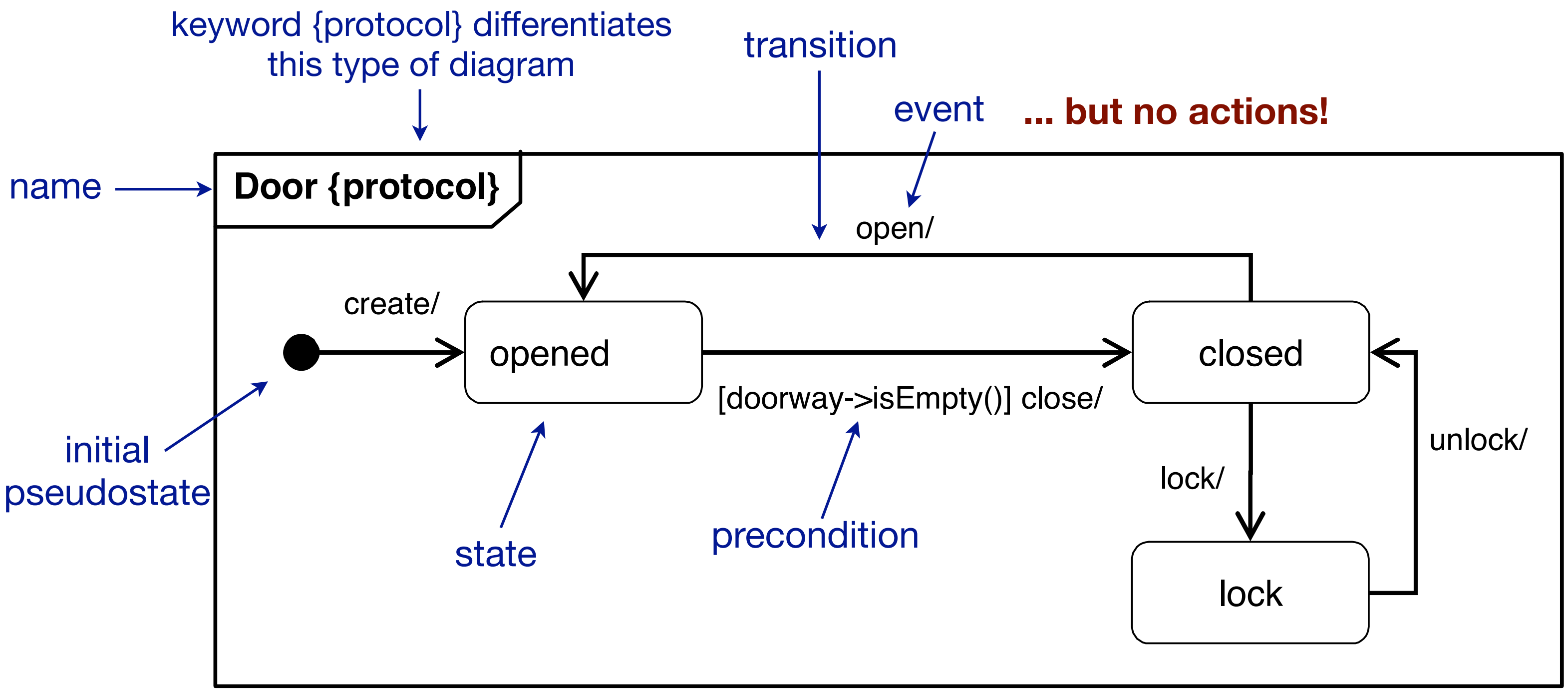
UML Protocol State Machine

Example:



[UML Superstructure Specification v2.2]

UML Protocol State Machine



[UML Superstructure Specification v2.2]

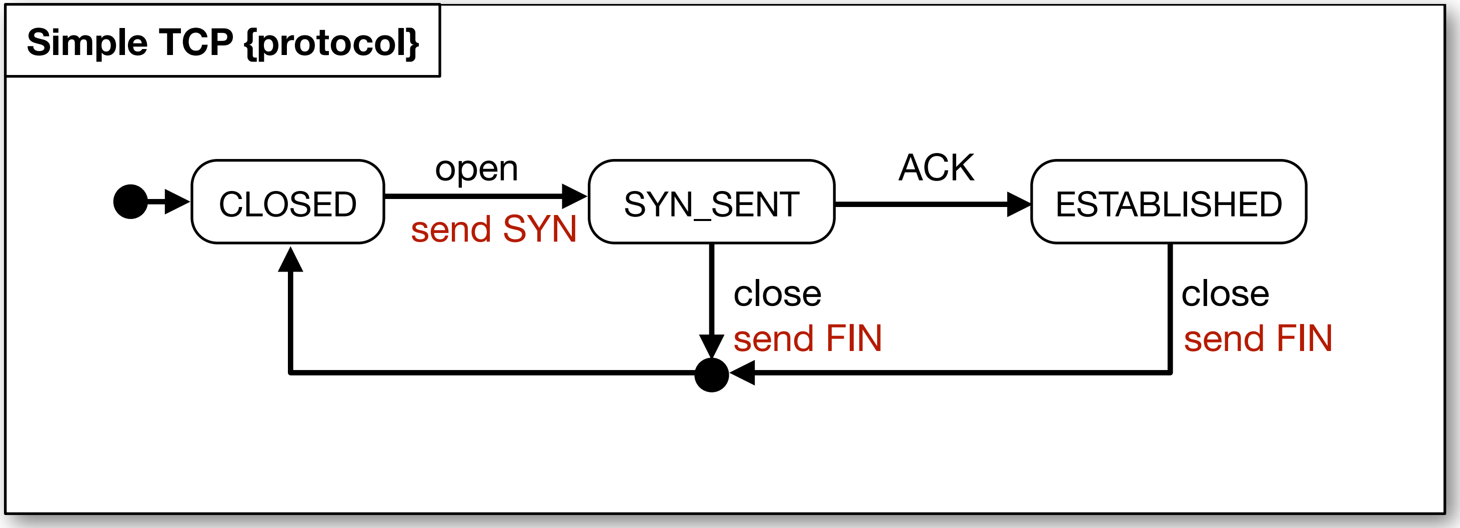
Protocol Transitions

- Notation of transitions:

$$\xrightarrow{\text{[precondition] event / [postcondition]}}$$

- Protocol transitions have **no associated actions**
  (in contrast to state machine transitions)

Example



Send signal actions are not modeled here.

Protocol state machines

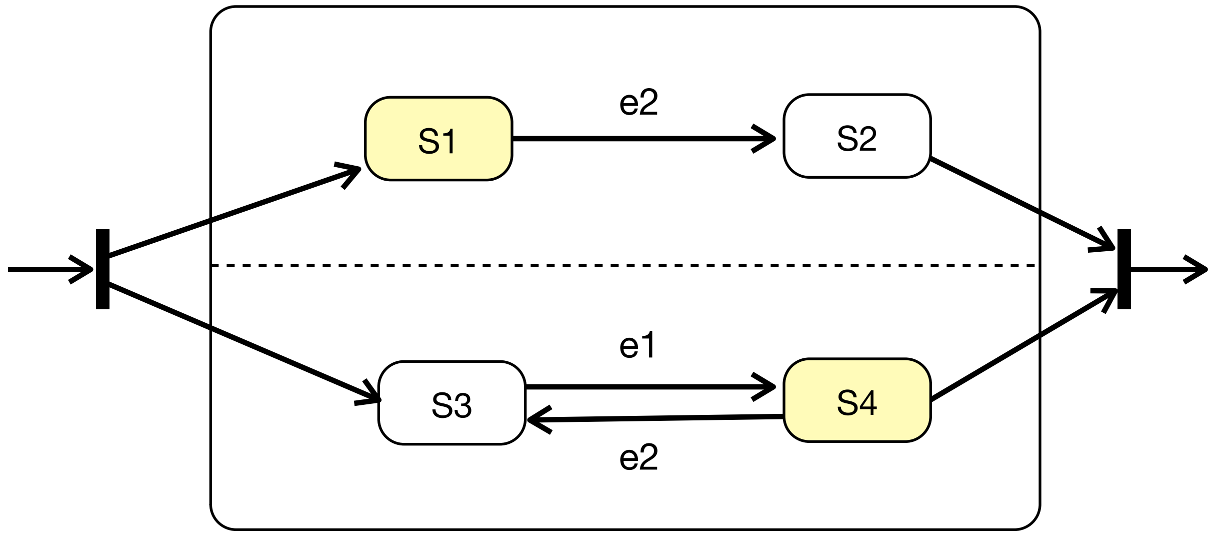
- Protocol state machines cannot describe responses such as sending acknowledgement messages
- Protocol state machines allow a **reflective** description of behaviour
- For a **constructive** description, state machines should be used

Semantic variation points

- Some UML elements have *semantic variation points*
- e.g. unexpected event reception (see UML Spec. 15.3.7)
  - What to do if there is a new message in the queue that cannot be handled?
    - ignore the event (delete the message)?
    - defer the event (leave the message in queue)?
    - raise an exception?
- e.g. concurrency: can two processes really be concurrent?
  - code generators enforce determinism

Semantic variation points

- Concurrency: Which transition is triggered first?

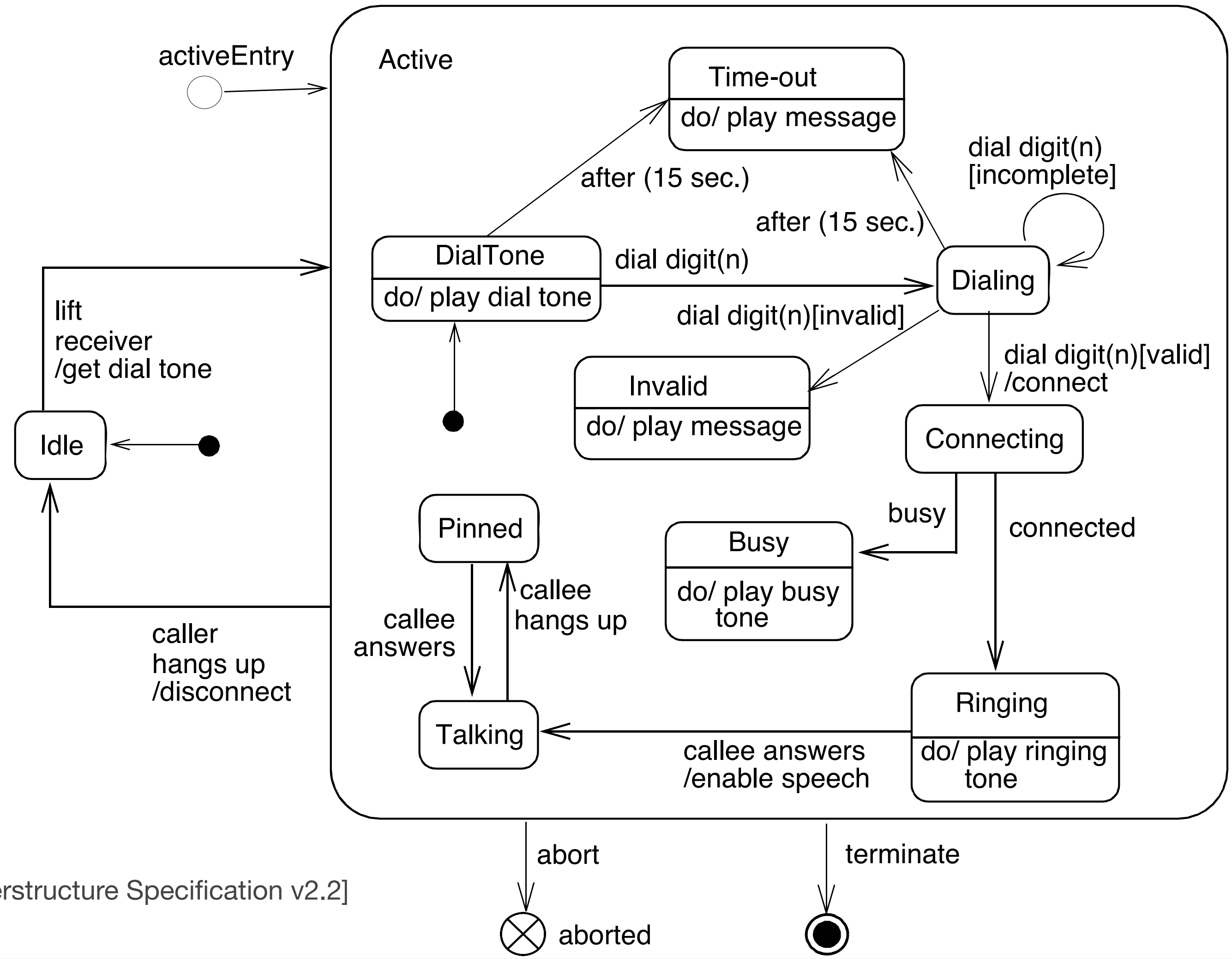


After event e1, states S1 and S4 are active.
Assume, e2 is the next event.

Modeling example (1)

- **Modeling a telephone:**
  1. play a dial tone after the caller lifts the receiver
  2. then allow the user to dial digits
     - quit after a timeout
     - quit after invalid digit
  3. establish connection
     - play busy tone if busy
     - play ringing tone otherwise
  4. enable talking until the caller hangs up

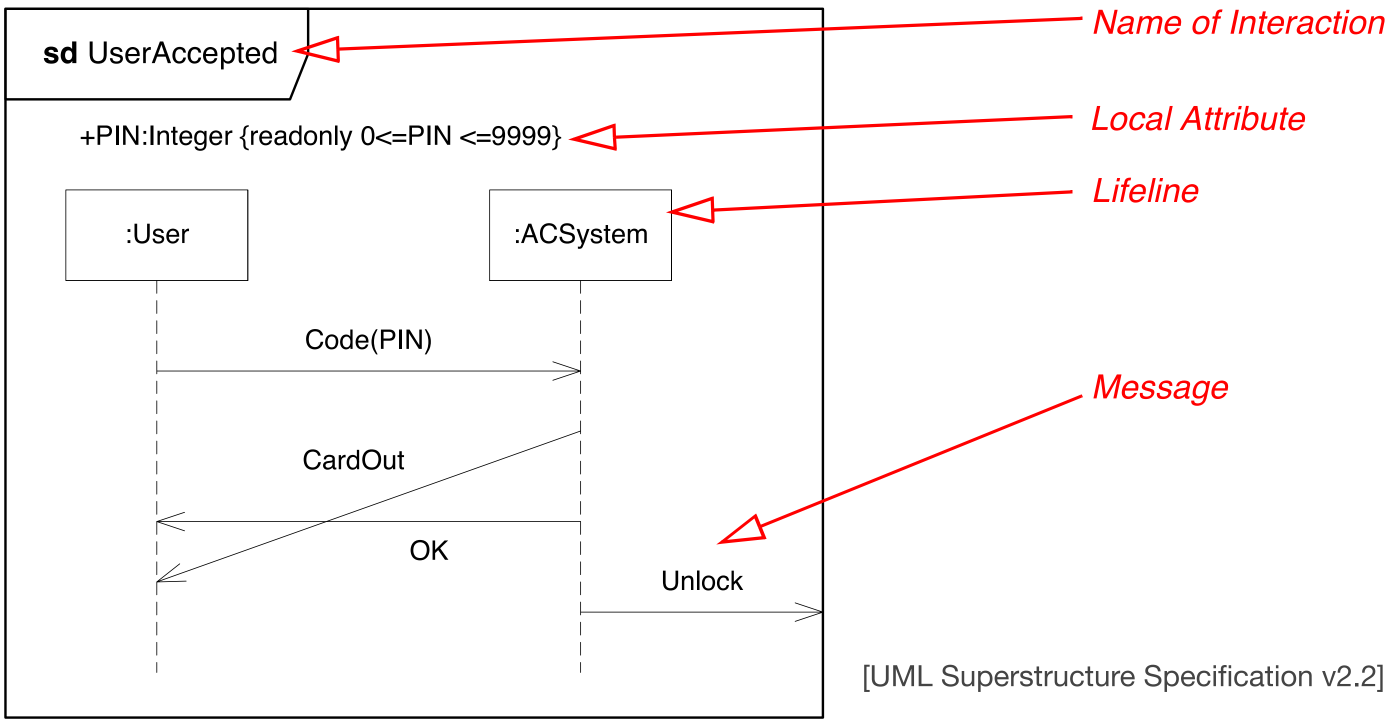
Modeling example (2)



[UML Superstructure Specification v2.2]

UML Sequence Diagrams

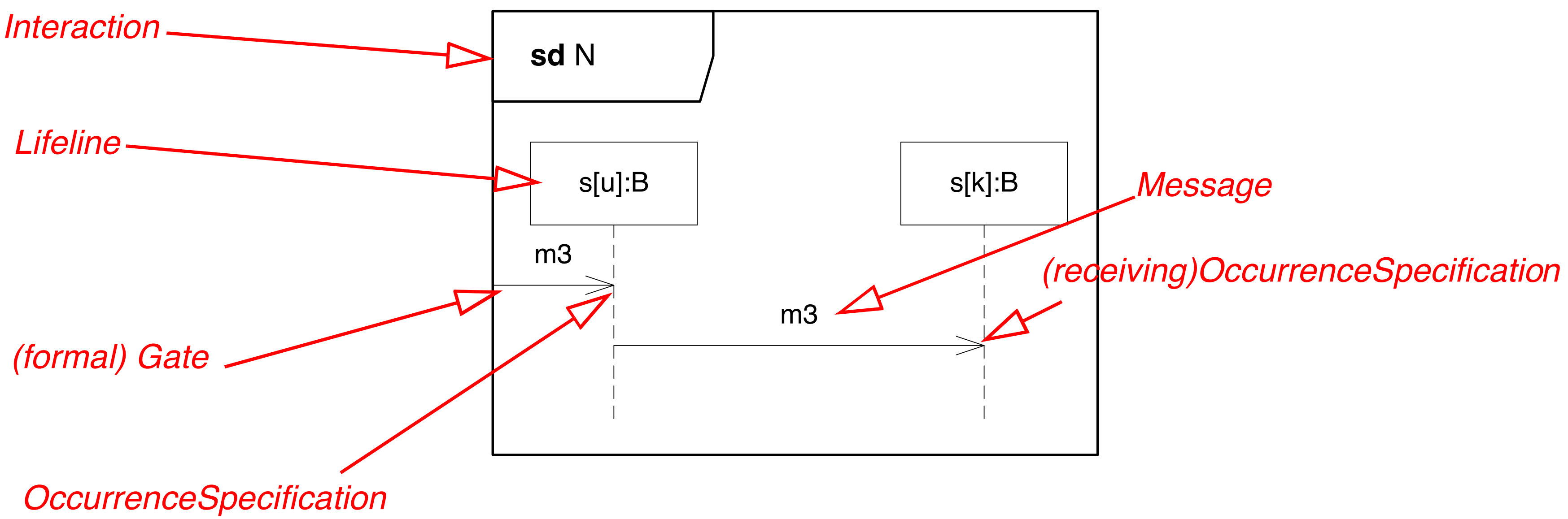
- Model process interaction (variant of interaction diagrams)
- Focus on message exchange
- Example:



[UML Superstructure Specification v2.2]

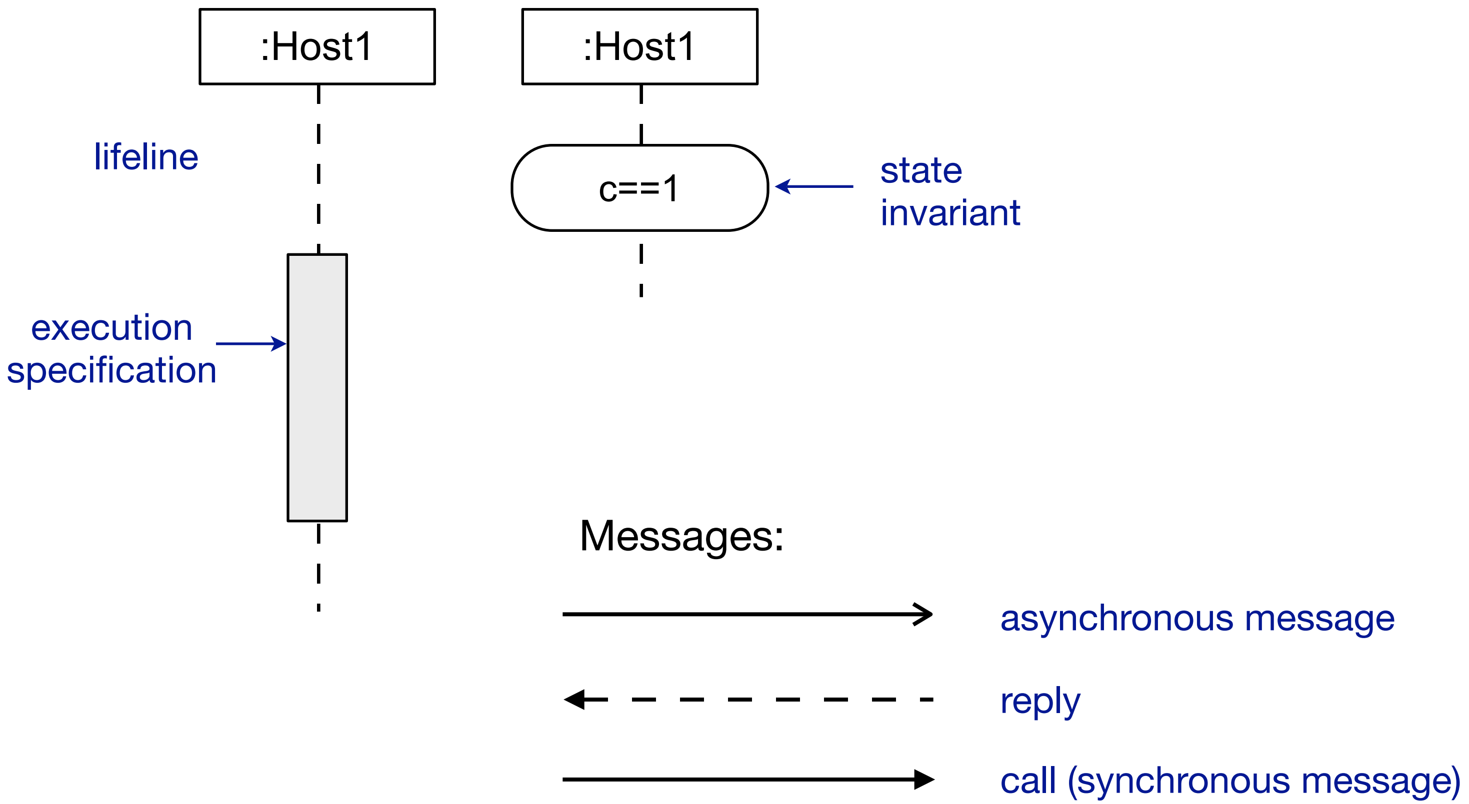
UML Sequence Diagrams

Example:

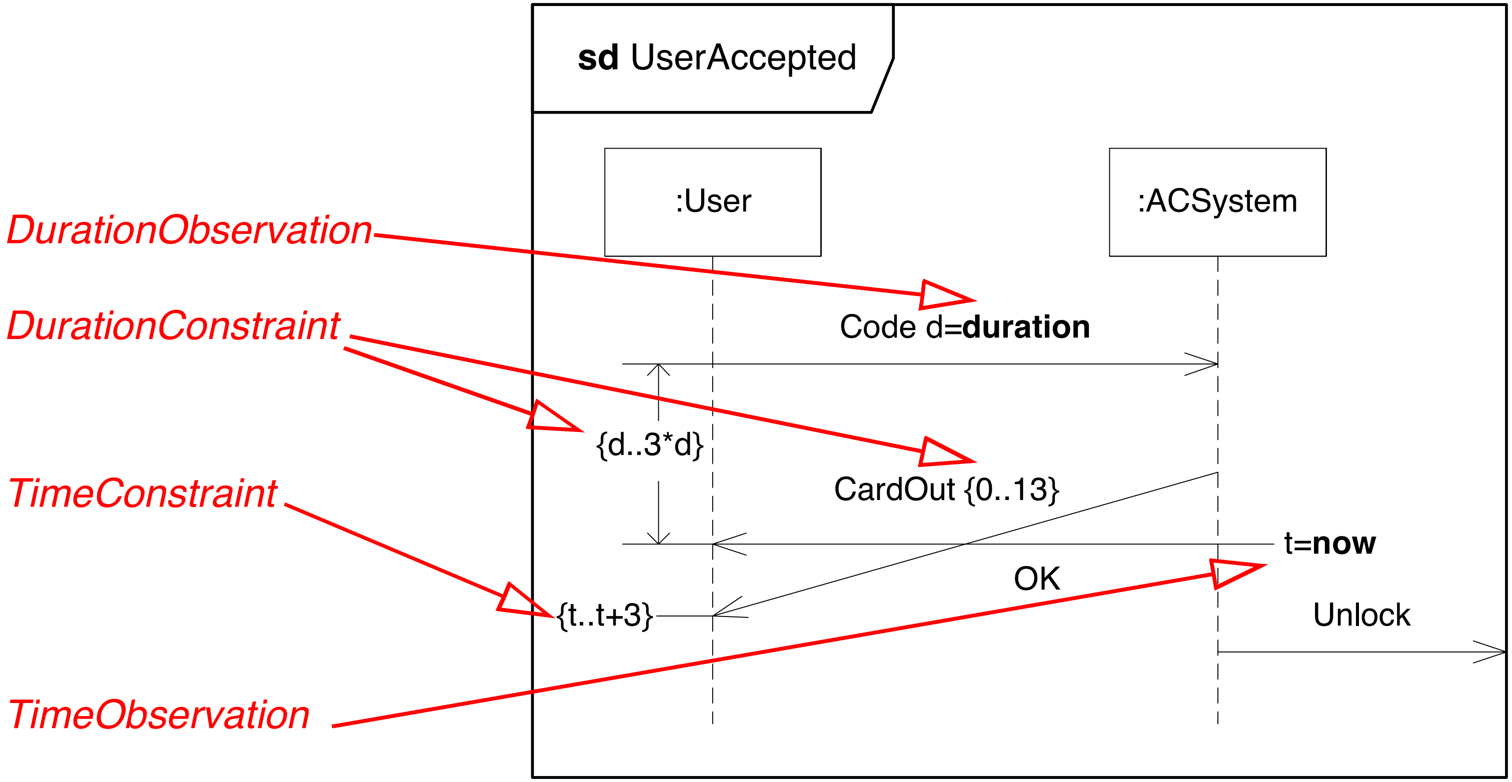


[UML Superstructure Specification v2.2]

Elements of Sequence Diagrams

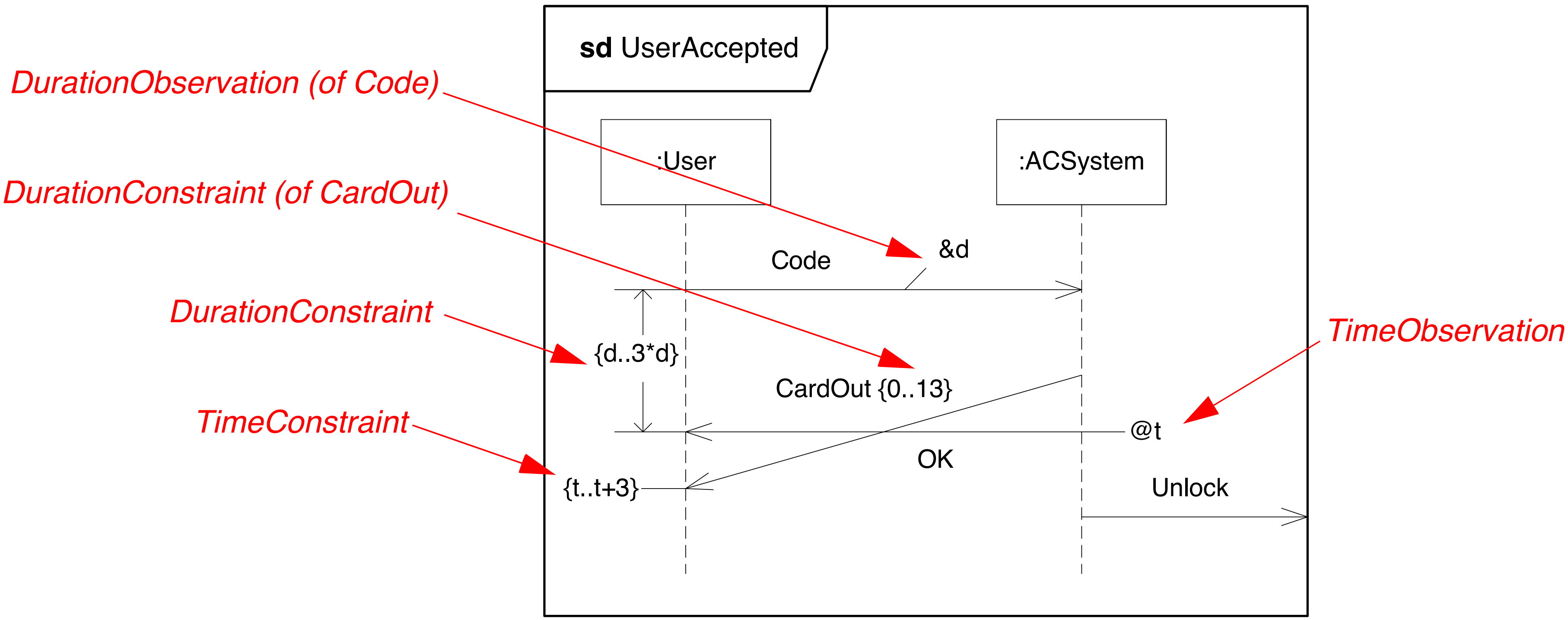


Sequence Diagram with Constraints (1)



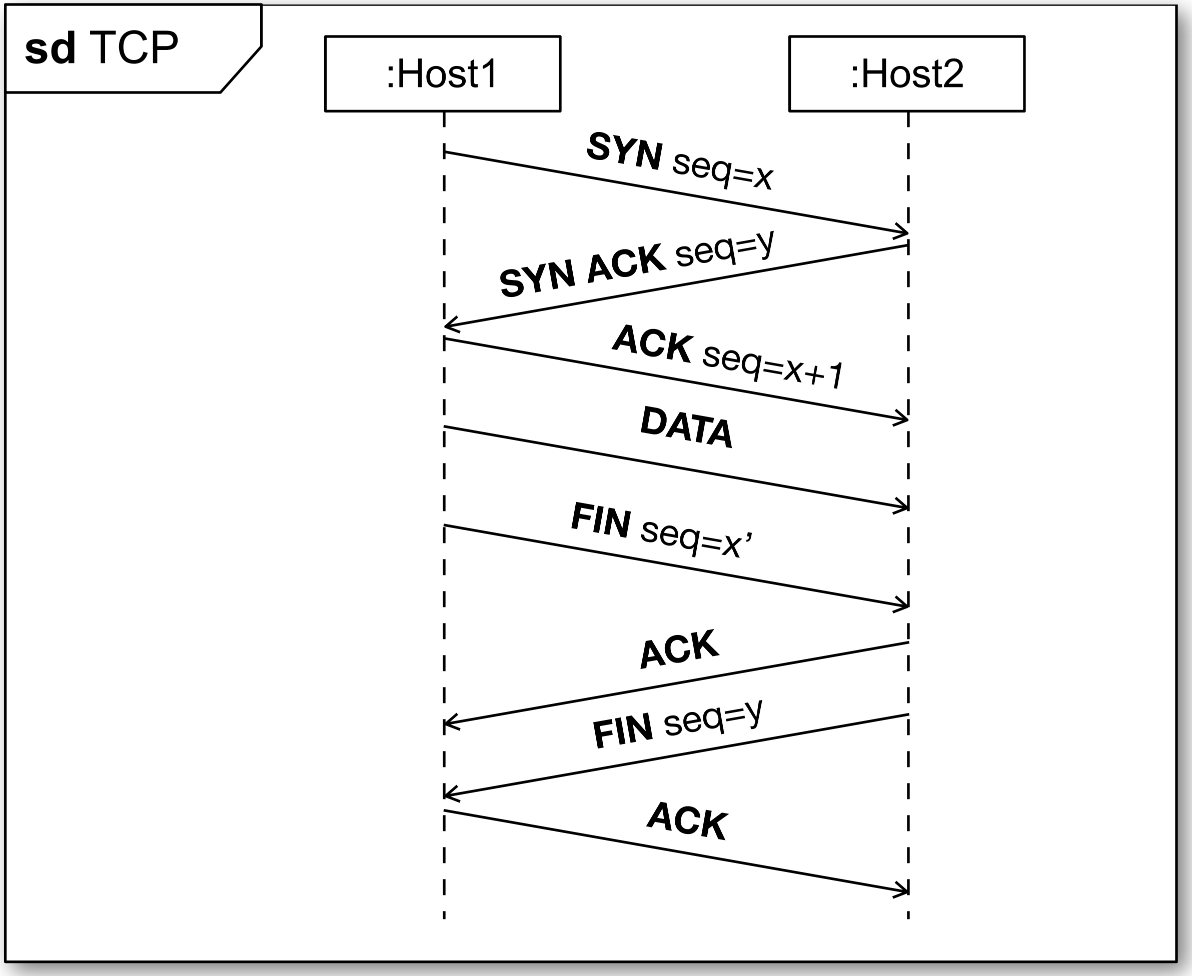
[UML Superstructure Specification v2.2]

Sequence Diagram with Constraints (2)



[UML Superstructure Specification v2.2]

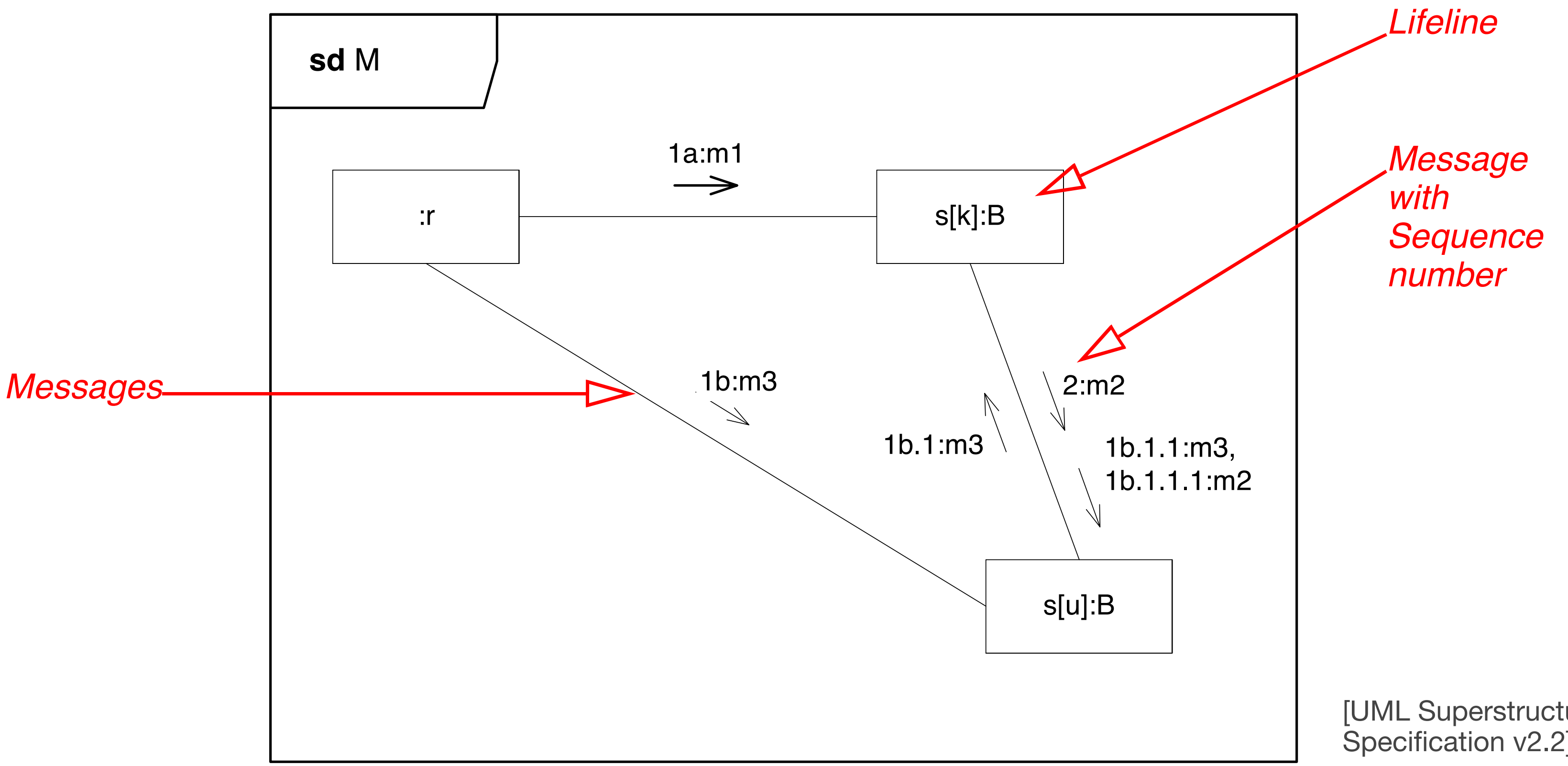
TCP Example



sd TCP
:Host1
:Host2
SYN seq=x
SYN ACK seq=y
ACK seq=x+1
DATA
FIN seq=x'
ACK
FIN seq=y
ACK

Communication Diagram

- Shows interactions from an architectural point of view



[UML Superstructure Specification v2.2]

UML Review

- Collection of diagrams and notations
- Semantics is not always clear (this is also a consequence of historical and political decisions)
- Useful for specification and documentation
- (Partly) supported by modeling tools
- Model-checking based on UML is still a research topic

more on semantics: lecture *Software Design, Modelling and Analysis in UML* by Bernd Westphal, Software Engineering workgroup, Uni Freiburg

UML Review

- UML state machines describe the behaviour in general (constructive description), used for
  - specification
  - documentation

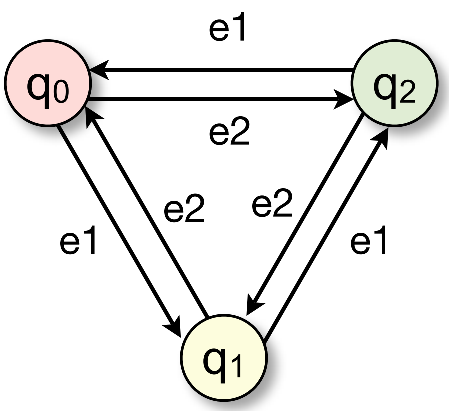
- UML sequence diagrams describe the specific behaviour during execution (reflective description), used for
  - describing test sequences
  - visualization of simulations
  - documentation

FSM Implementation

- Generic techniques (for C++, Java, ...):
  - The nested switch/case technique
    - define a switch for the states, in each state define a switch for events
    - change of behaviour by conditional statements
  - The State Design Pattern
    - define an abstract superclass with an event handler and derive a concrete class for each state
    - associate the state with the class holding the context (the state machine)
    - change of behaviour by object change

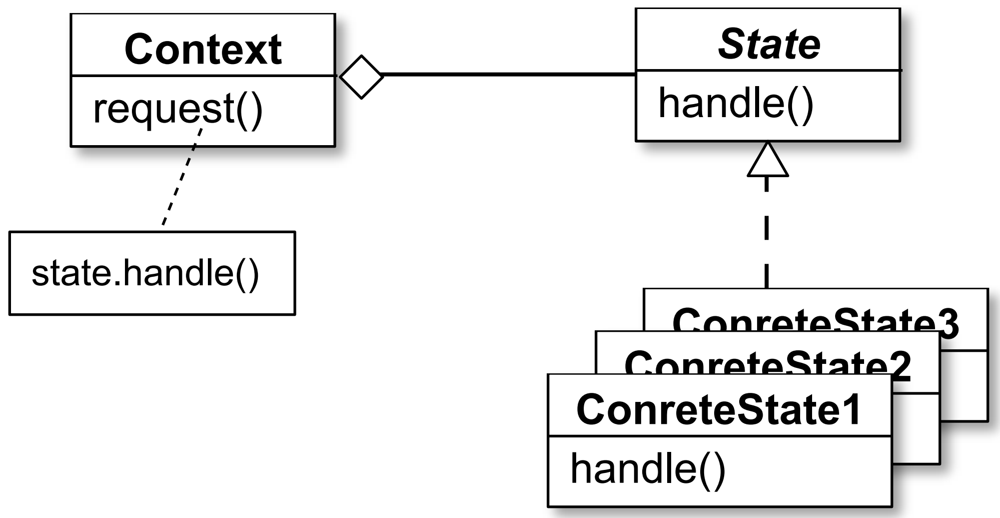
Nested switch/case

```
enum State {q0, q1, q2, ...};
enum Event {e1, e2, ...};

static State s = q0;

void handle(Event e)
{
  switch(s)
  {
  case q0:
    switch(e)
    {
    case e1:
      s = q1;
    break;
    case e2:
      s = q2;
    break;
    [...]
    }
  break;

  case q1:
    switch(e)
    {
    case e1:
      s = q2;
    break;
    case e2:
      s = q0;
    break;
    [...]
    }
  break;

  case q2:
    switch(e)
    {
    case e1:
      s = q0;
    break;
    case e2:
      s = q1;
    break;
    [...]
    }
  break;
  [...]
  }
}
```



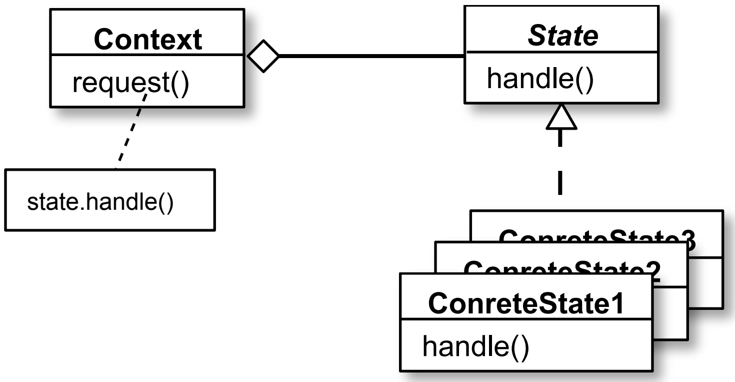
State pattern

- Separate classes for different states
- State change by instantiating a new object



State pattern

```
class Context {
  private State state;
  public void setState(State s) {
    state = s;
  }
  handleEvent(Event e) {
    state.handle(e, this);
  }
}

interface State {
  public void handle(Event e, Context c)
}

class ConcreteState1 implements State {
  public void handle(Event e, Context c) {
    switch (e)
    case e1: context.setState(new State1); break;
    case e2: context.setState(new State2); break;
  }
}

class ConcreteState2 implements State {
  [...]
}

[...]
```

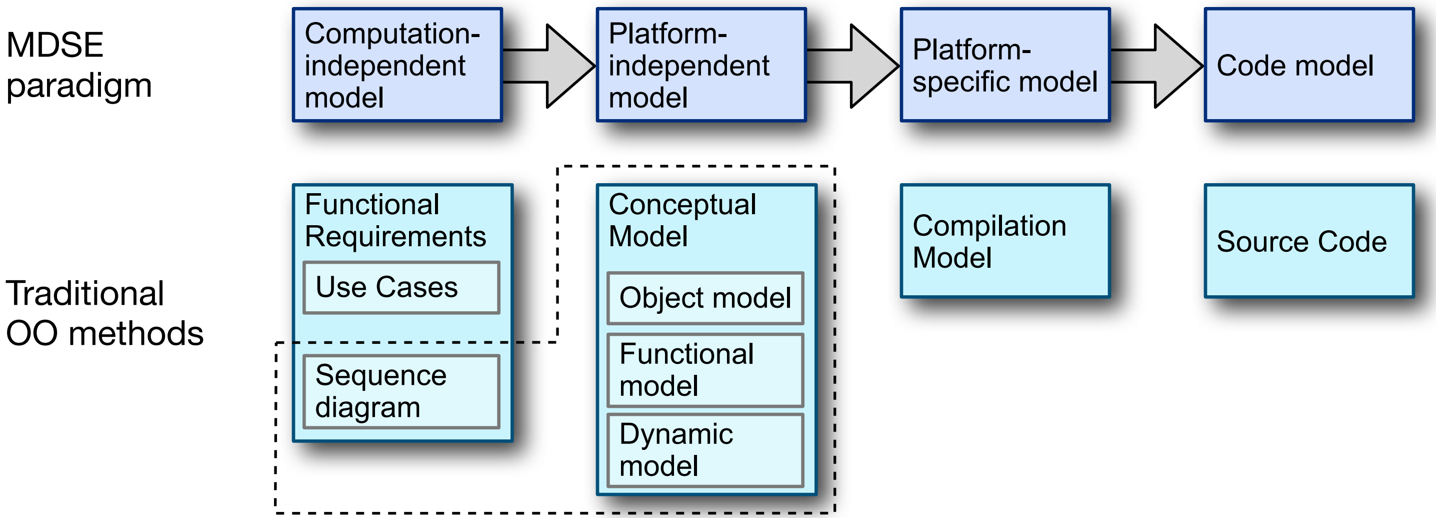


FSM Implementation

- Nested switch/case
  - suitable for small number of states and events with only few actions
  - **hopefully you don't need to program and maintain this by hand...**
- State design pattern
  - generally better maintainable
  - oversized for small state machines
  - state classes can be tested separately

Automatic code generation

- Code generation from state charts
- Used in Model-driven Software Engineering



The Model-driven Software Architecture paradigm

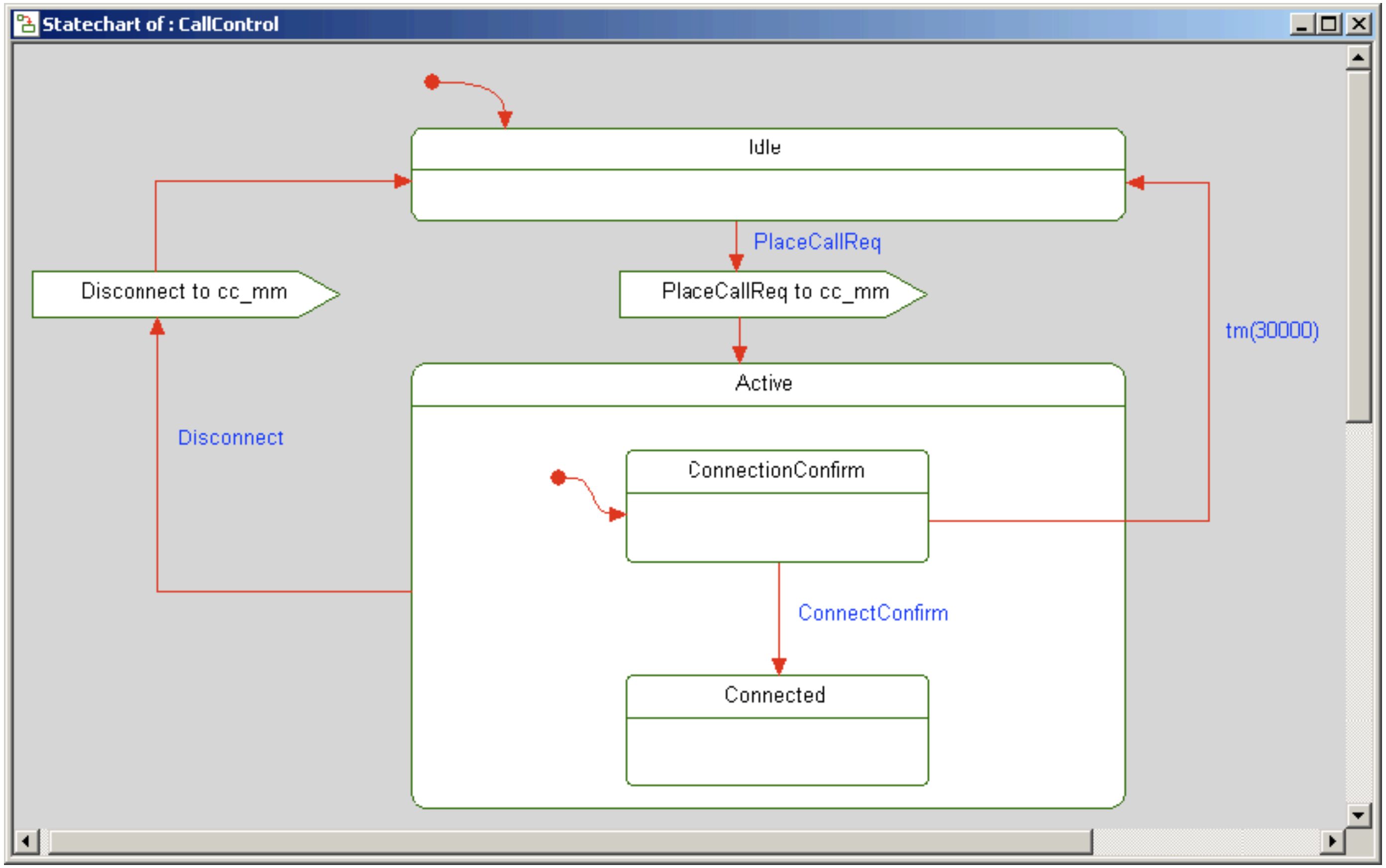
cf. [Pastor et al.: "Model-driven Development", Informatik Spektrum 31(5), 2008]

Automatic code generation

- Code generation from state charts can be performed by tools for Model-Driven Software Engineering (MDSE), e.g. IBM/Telelogic Rhapsody
  - Graphical UML state machine modeling
  - C++/Java code generation
  - Simulation and animation
    (special instructions inserted into the code)
  - Simulation run can be shown in a sequence diagram

State machines in MDSE

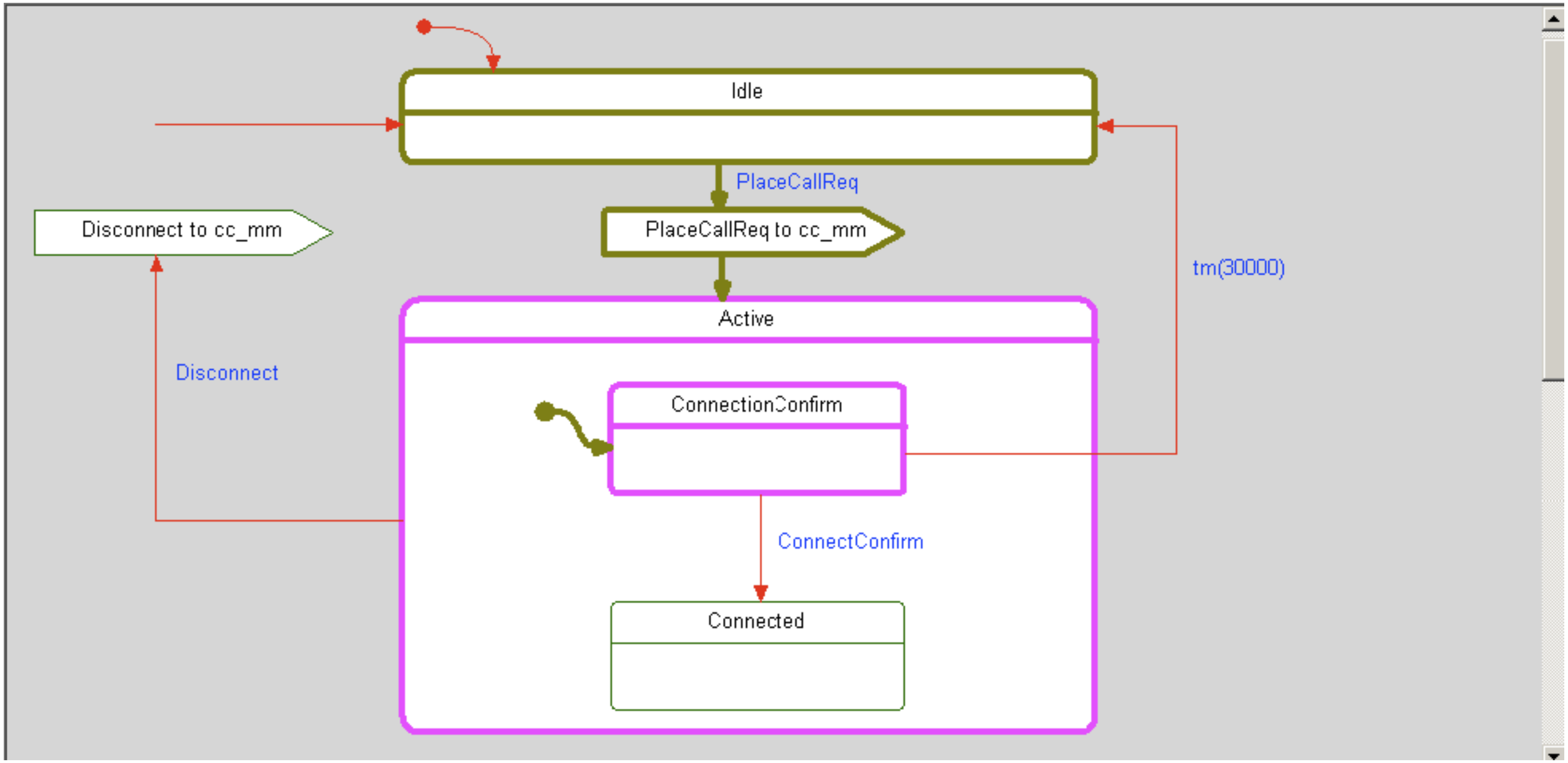
Modeling state charts with Rhapsody®



[IBM/Telelogic Rhapsody 7.4 Tutorial, 2008]

State machines in MDSE

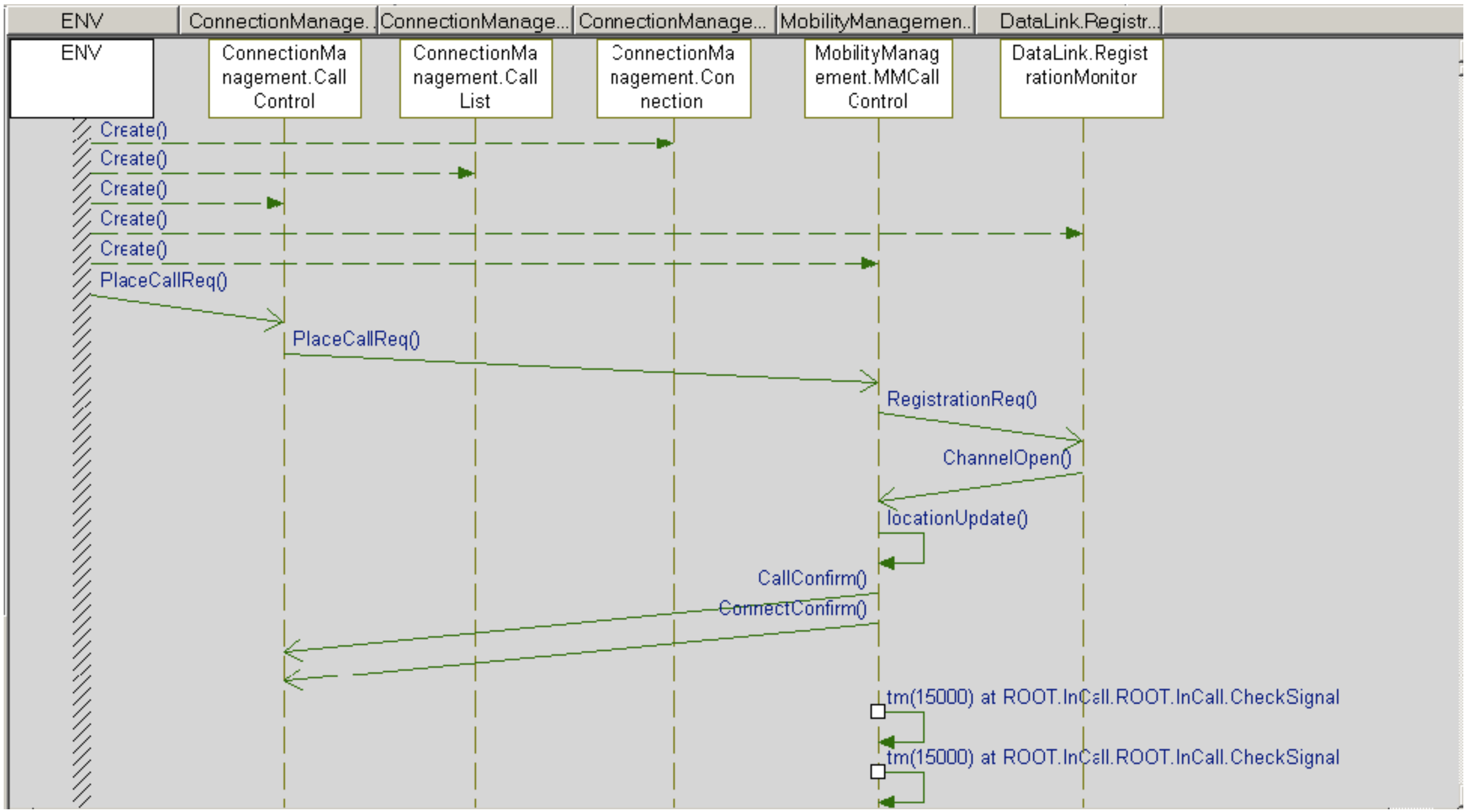
State chart animation with Rhapsody®



[IBM/Telelogic Rhapsody 7.4 Tutorial, 2008]

State machines in MDSE

Sequence diagram from state chart animation with Rhapsody®



[IBM/Telelogic Rhapsody 7.4 Tutorial, 2008]