Run-to-Completion Step (1)

$$
\begin{array}{l}
\text{RTC}(\mathit{env}, \mathit{conf}) \equiv \\
\quad \lceil \mathit{event} \leftarrow \texttt{fetch}() \\
\qquad \mathit{step} \leftarrow \text{choose steps}(\mathit{conf}, \mathit{event}) \\
\qquad \text{if } \mathit{step} = \emptyset \wedge \mathit{event} \in \mathit{deferred}(\mathit{conf}) \\
\qquad \text{then } \texttt{defer}(\mathit{event}) \\
\qquad \text{fi} \\
\qquad \text{for } \mathit{transition} \in \mathit{step} \text{ do} \\
\qquad \quad \mathit{conf} \leftarrow \text{handleTransition}(\mathit{env}, \mathit{conf}, \mathit{transition}) \\
\qquad \text{od} \\
\qquad \text{if } \mathit{isCall}(\mathit{event}) \wedge \mathit{event} \notin \mathit{deferred}(\mathit{conf}) \\
\qquad \text{then } \texttt{acknowledge}(\mathit{event}) \\
\qquad \text{fi} \\
\qquad \mathit{conf} \rfloor
\end{array}
$$

Run-to-Completion Step (2)

steps(env, $conf$, $event$) $\equiv$
$\lceil transitions \leftarrow enabled(env, conf, event)$
$\{step \mid (guard, step) \in$ steps$(conf, transitions) \wedge env \models guard \} \rfloor$

steps($conf$, $transitions$) $\equiv$
$\lceil steps \leftarrow$ {(true, $\emptyset$)}
for $transition \in transitions$ do
for $(guard, step) \in$ steps$(conf, transitions \setminus \{transition\})$ do
if $inConflict(conf, transition, step)$
then if $higherPriority(conf, transition, step)$
then $guard \leftarrow guard \wedge \neg guard(transition)$ fi
else $step \leftarrow step \cup \{transition\}$
$guard \leftarrow guard \wedge guard(transition)$ fi
$steps \leftarrow steps \cup \{(guard, step)\}$ od od
$steps \rfloor$

Run-to-Completion Step (3)

```
handleTransition(conf, transition) ≡
  ⌈for state ∈ insideOut(exited(transition)) do
      uncomplete(state)
      for timer ∈ timers(state) do stopTimer(timer) od
      execute(exit(state))
      conf ← conf \ {state}
   od
   execute(effect(transition))
   for state ∈ outsideIn(entered(transition)) do
      execute(entry(state))
      for timer ∈ timers(state) do startTimer(timer) od
      conf ← conf ∪ {state}
      complete(conf, state)
   od
   conf⌋
```

Semantic variation points

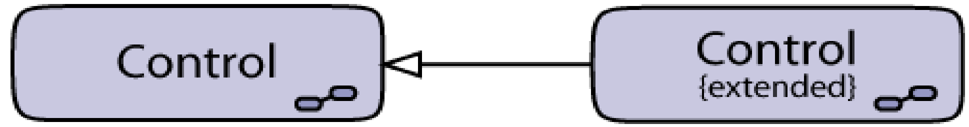
- Some semantic variation points have been mentioned before.
  - delays in event pool
  - handling of deferred events
  - entering of composite states without default entry

- Which events are prioritized?
  - completion events only
  - all internal events (completion, time, change)

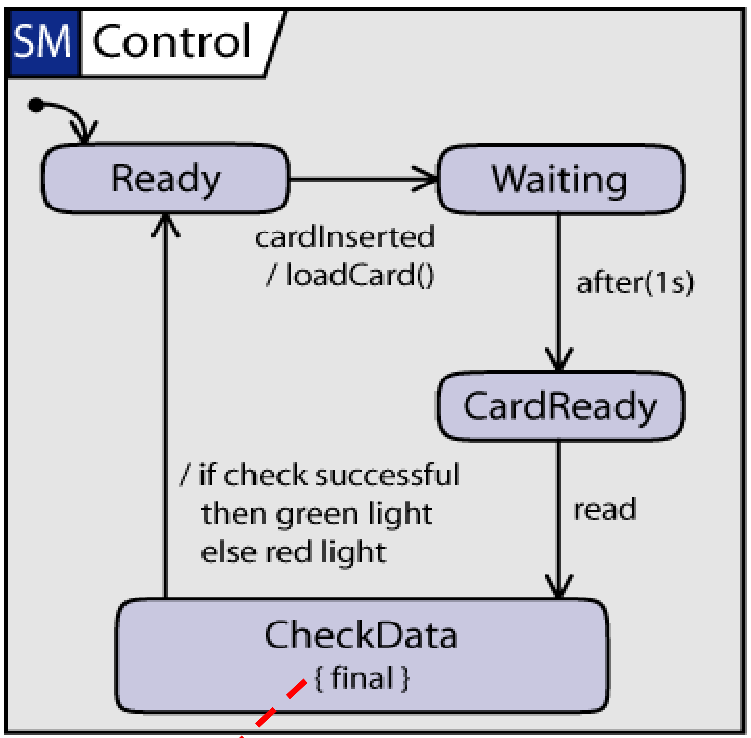
- Which (additional) timing assumptions?
  - delays in communication
  - time for run-to-completion step
    - zero-time assumption

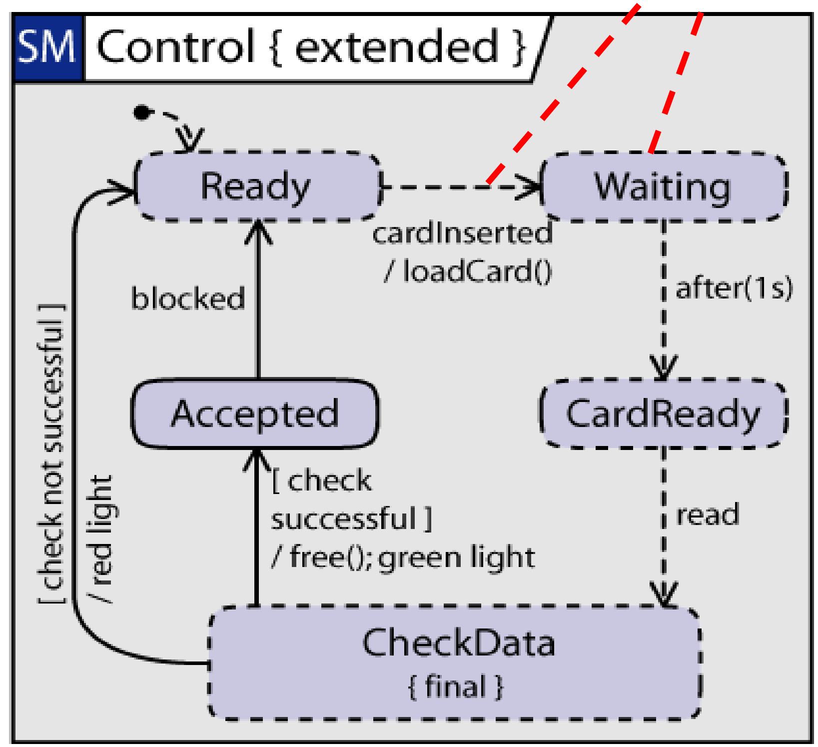
State machine refinement

- State machines are behaviors and may thus be refined.



not refined (may be omitted)

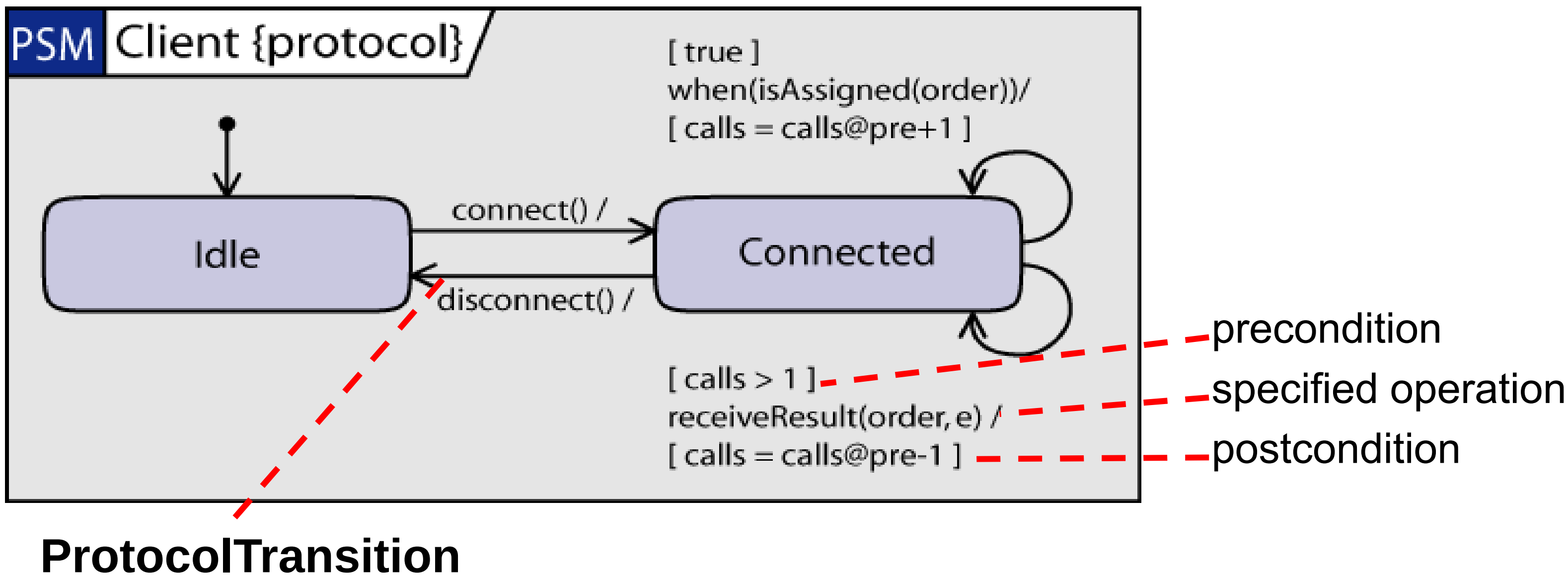




no refinement possible

Protocol state machines

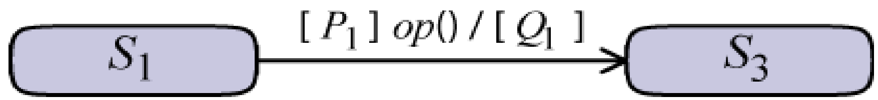
- Protocol state machines specify which behavioral features of a classifier can be called in which state and under which condition and what effects are expected.
  - particularly useful for object life cycles and ports
  - no effects on transitions, only effect descriptions



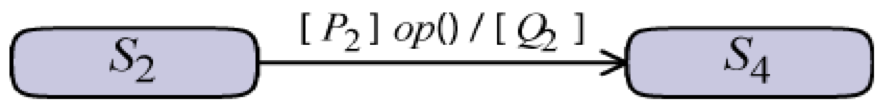
Protocol state machines

Several operation specifications are combined conjunctively:

context C::op()
pre: inState(S_1) **and** P_1
post: Q_1 **and** inState(S_3)



context C::op()
pre: inState(S_2) **and** P_2
post: Q_2 **and** inState(S_4)



results in

context C::op()
pre: (inState(S_1) **and** P_1) **or** (inState(S_2) **and** P_2)
post: (inState@pre(S_1) **and** P_1@pre) **implies** (Q_1 **and** inState(S_3))
and (inState@pre(S_2) **and** P_2@pre) **implies** (Q_2 **and** inState(S_4))

How things work together

- Static structure
  - sets the scene for state machine behavior
  - state machines refer to
    - properties
    - behavioral features (operations, receptions)
    - signals
- Interactions
  - may be used to exemplify the communication of state machines
  - refer to event occurrences used in state machines
- OCL
  - may be used to specify guards and pre-/post-conditions
  - refers to actions of state machines (`OclMessage`)
- Protocols and components
  - state machines may specify protocol roles

Wrap up

- State machines model behaviour
  - object and use case life cycles
  - control automata
  - protocols

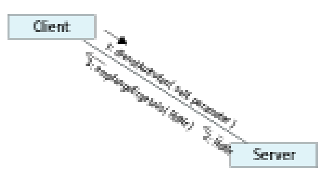
- State machines consist of
  - Regions and …
  - … (Pseudo)States (with entry, exit, and do-activities) …
  - connected by Transitions (with triggers, guards, and effects)

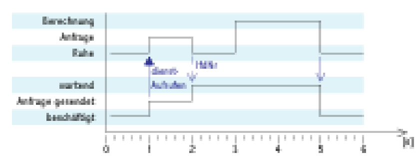
- State machines communicate via event pools.

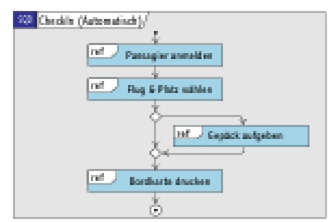
- State machines are executed by run-to-completion steps.

Unified Modeling Language 2

Interactions





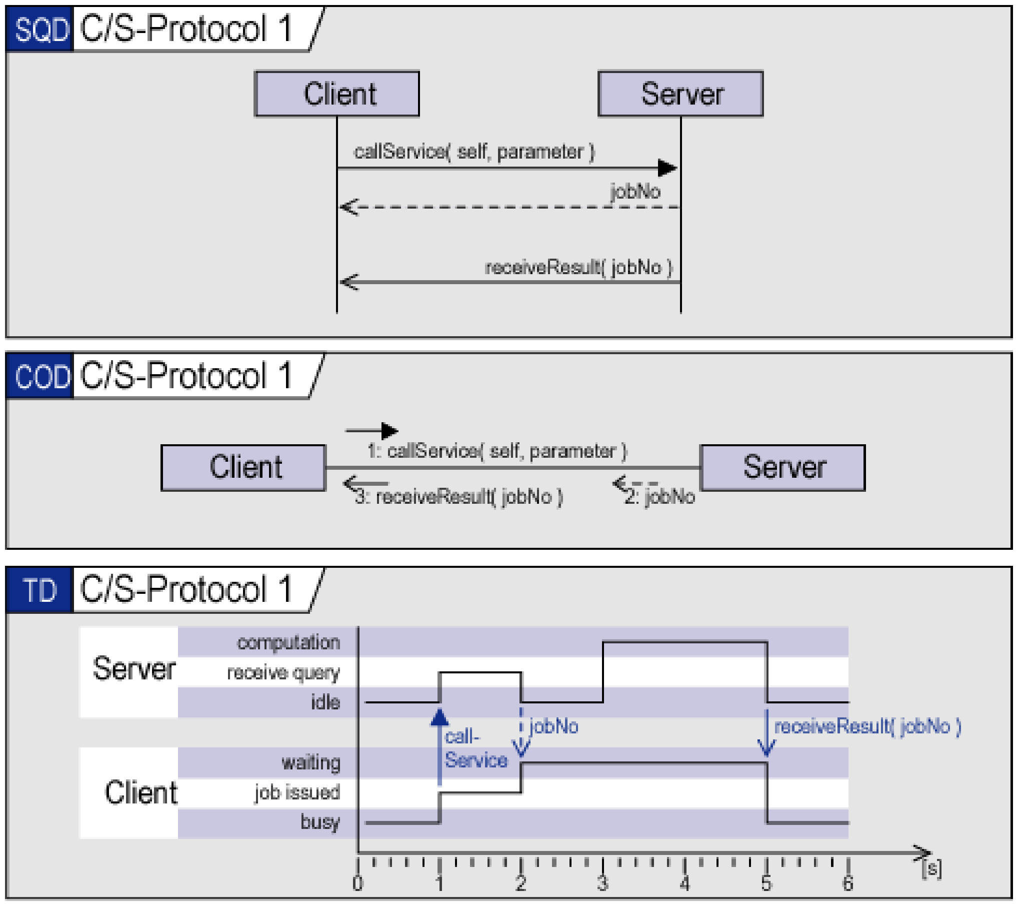


A first glimpse

sequence diagram

communication diagram

timing diagram



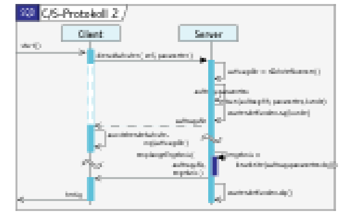
all three are semantically equivalent

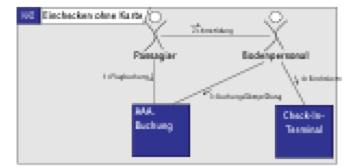
History and predecessors

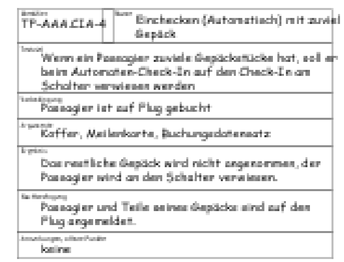
- Simple sequence diagrams
  - 1990's
    - Message Sequence Charts (MSCs) used in TelCo-industry
    - several OO-methods use sequence diagrams
- Complex sequence diagrams
  - 1996: Complex MSCs introduced in standard MSC96
  - 1999: Life Sequence Charts (LSCs)
- Communication diagrams
  - 1991: used in Booch method
  - 1994: used in Cook/Daniels: Syntropy
- Timing diagrams
  - traditionally used in electrical engineering
  - 1991: used in Booch method
  - 1993: used in early MSCs
- Interaction overview
  - 1996: high-level MSCs (graphs of MSCs as notational alternative)

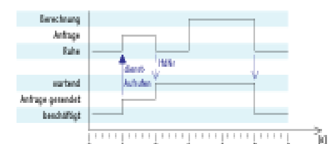
Usage scenarios

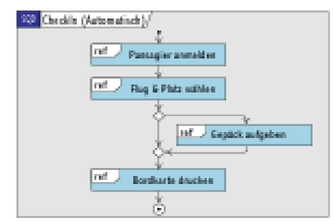
- Class/object interactions
  - design or document message exchange between objects
  - express synchronous/asynchronous messages, signals and calls, activation, timing constraints
- Use case scenarios
  - illustrate a use case by concrete scenario
  - useful in design/documentation of business processes (i.e. analysis phase and reengineering)
- Test cases
  - describe test cases on all abstraction levels
- Timing specification/documentation
- Interaction overview
  - organize a large number of interactions in a more visual style
  - defined as equivalent to using interaction operators





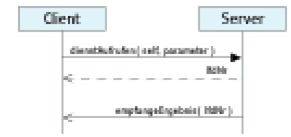




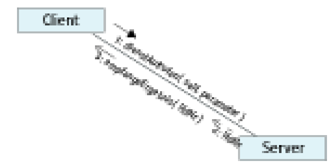


Syntactical variants

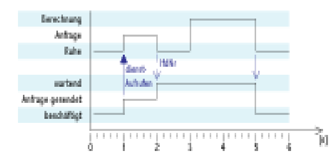
- Sequence diagram
  - traditional sequence diagrams + interaction operators
  - focuses on exchanging many messages in complex patterns among few interaction partners



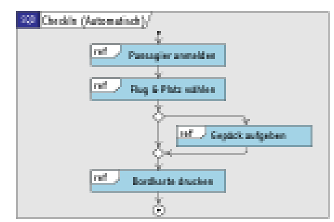
- Communication diagram
  - “collaboration diagram” in UML 1.x
  - focuses on exchanging few messages between (many) interaction partners in complex configuration



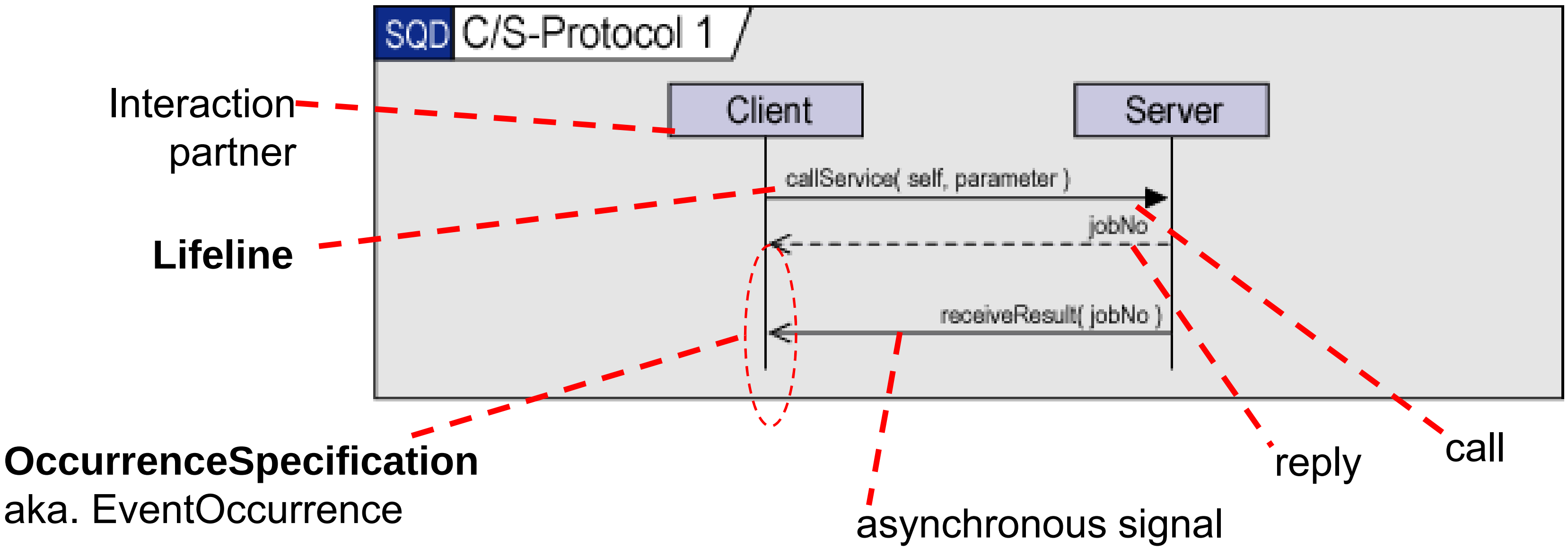
- Timing diagram
  - new in UML 2.0, oscilloscope-type representation, not necessarily metric time
  - focuses on (real) time and coordinated state change of interaction partners over time



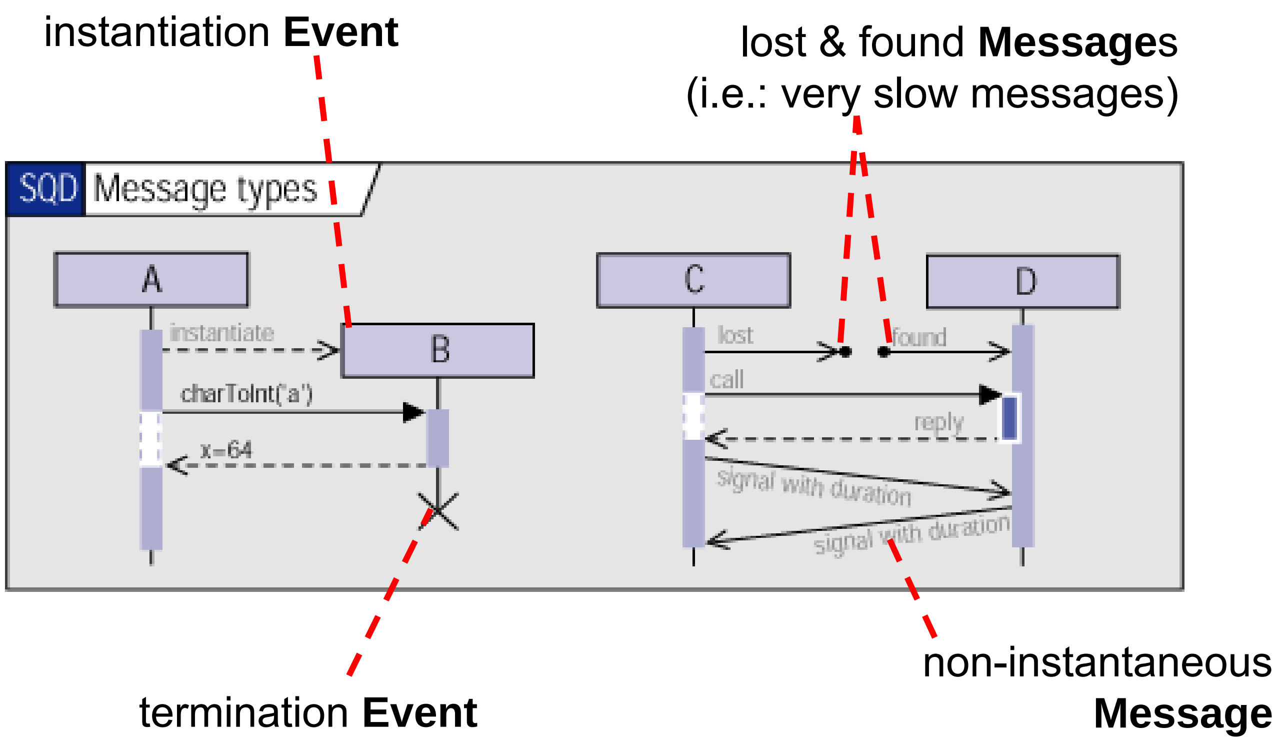
- Interaction overview diagram
  - looks like restricted activity diagram, but isn’t
  - arrange elementary interactions to highlight their interaction



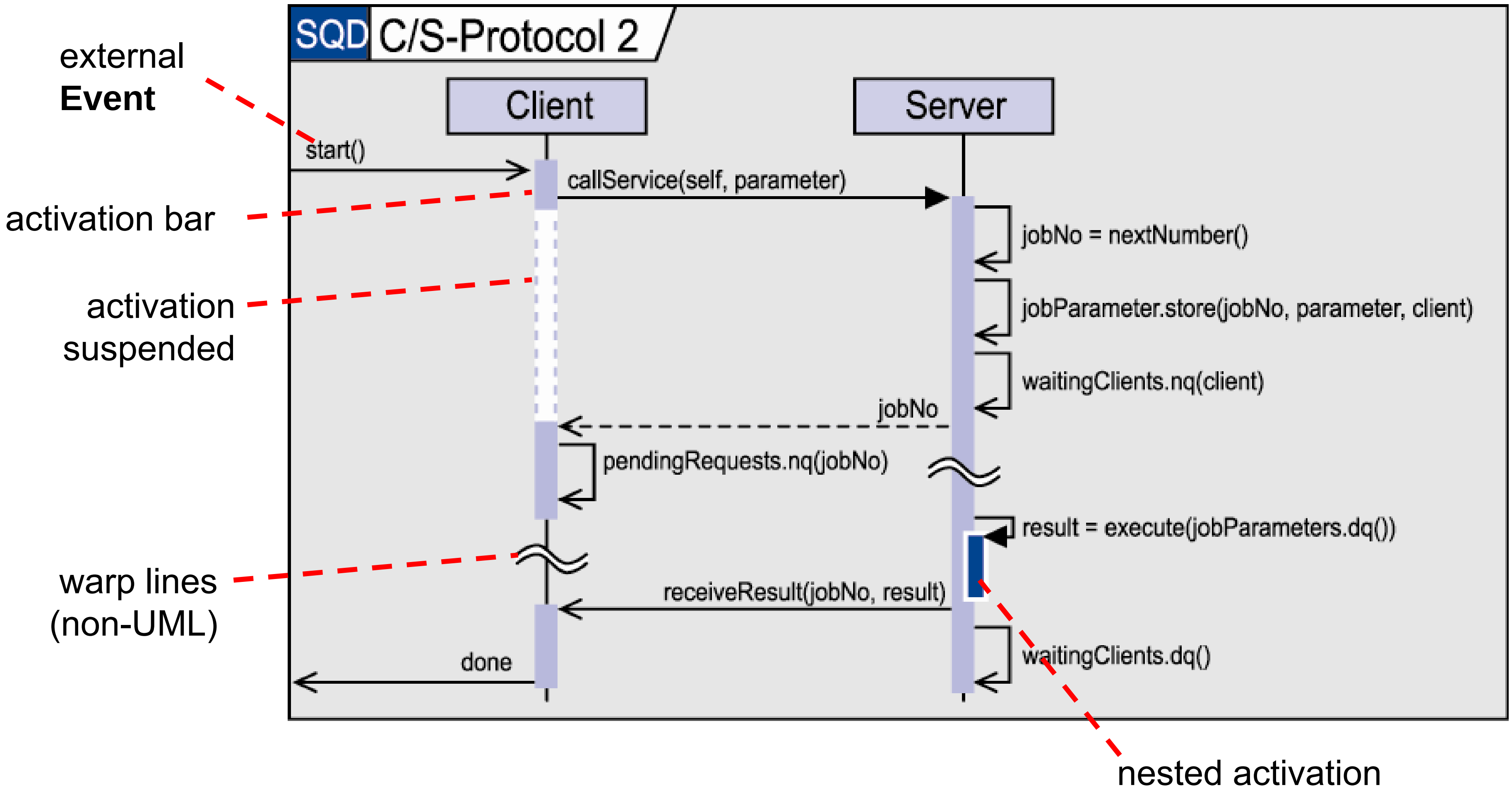
Main concepts



Message types

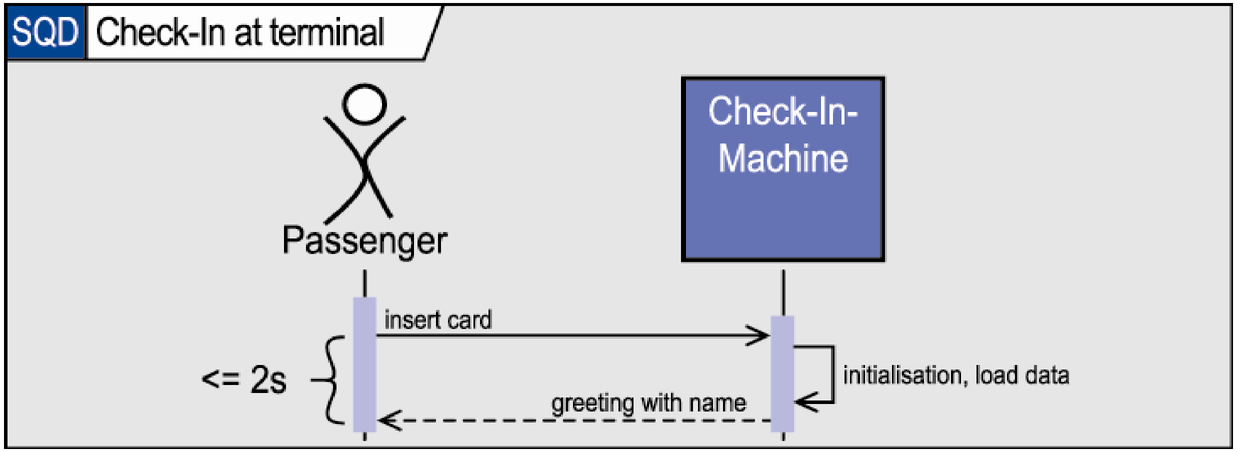


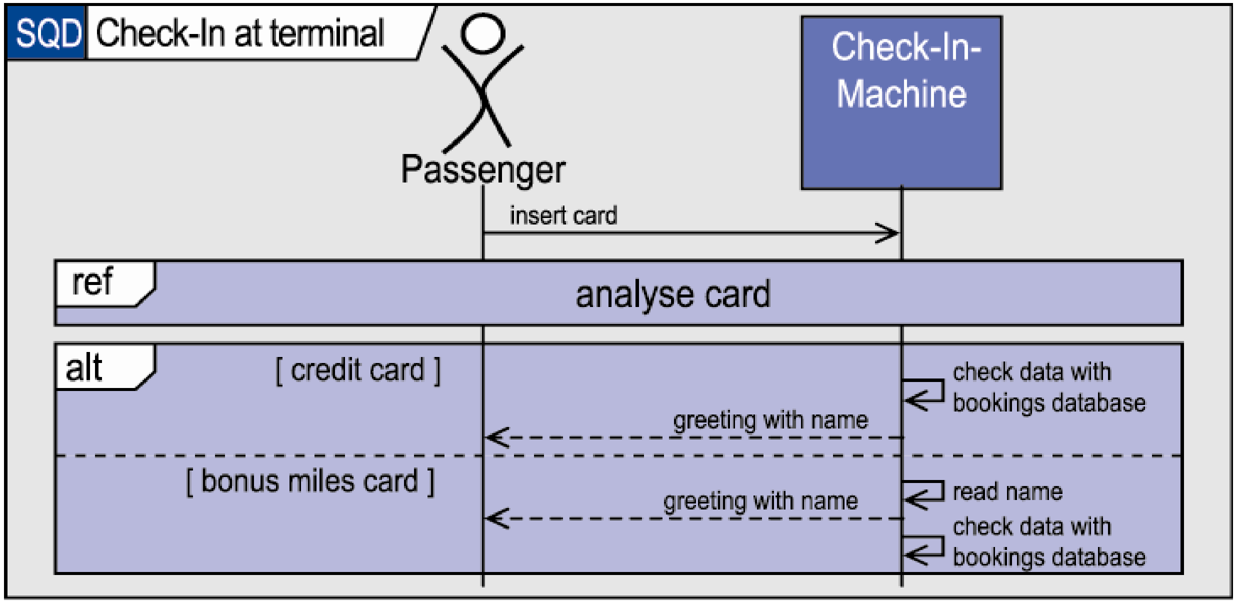
Activation


external
Event
activation bar
activation
suspended
warp lines
(non-UML)
nested activation
SQD C/S-Protocol 2
Client
Server
start()
callService(self, parameter)
jobNo = nextNumber()
jobParameter.store(jobNo, parameter, client)
waitingClients.nq(client)
jobNo
pendingRequests.nq(jobNo)
result = execute(jobParameters.dq())
receiveResult(jobNo, result)
waitingClients.dq()
done

Usage: Use case scenarios

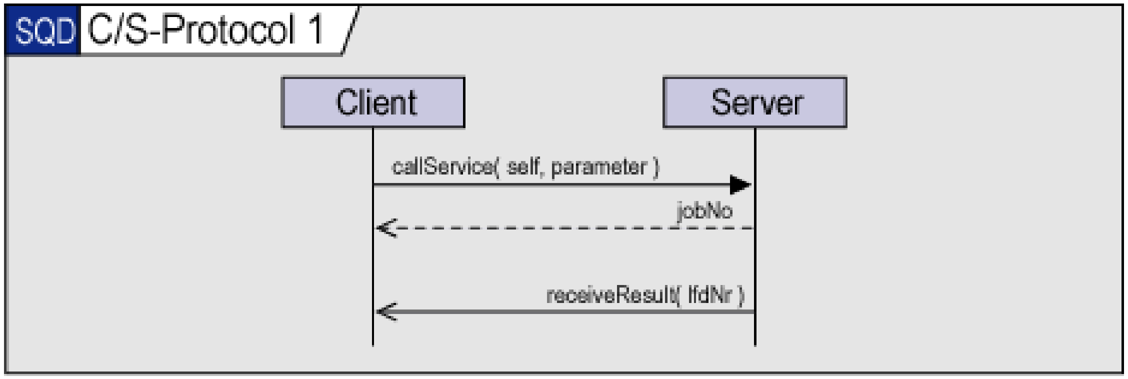
- Interaction **participants** are actors and systems rather than classes and objects.
- May be **refined** successively.
- Useful also for specifying high-level non-functional requirements such as response times.
- All kinds of interaction diagrams may be applied, depending on the circumstances.

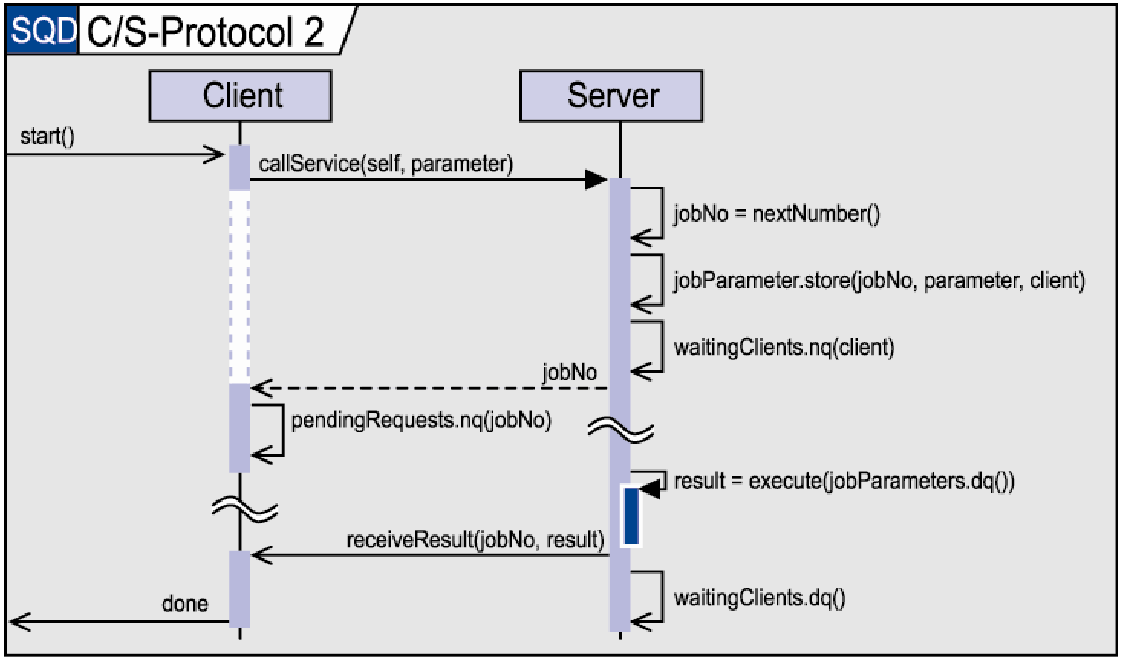




Usage: Class interactions

- Interaction **participants** are classes and objects rather than actors and systems.
- Again, successive **refinement** may be applied in different styles:
  - break down processing of messages
  - break down structure of interaction participants.
- All kinds of interaction diagrams may be applied, depending on the circumstances.





Usage: Test cases

- Like any other interaction, but with a different intention.
- Typically accompanied by a **tabular description** of purpose, expected parameters and result (similar to use case description).

identifier	name
TF-AAA.CIA-4	Check In (automatic) too much luggage
test goal	If a passenger has too many pieces of luggage and tries to check in using the check in machine, he should be referred to the check in counter.
precondition	passenger is booked on respective flight
arguments	luggage, bonus mile card, booking data
result	passenger is referred to counter
postcondition	luggage is not checked in, passenger is checked in
remarks, open questions	none

Usage: Timing specification

- For **embedded** and **real-time** systems, it may be important to specify absolute timings and state evolution over time.
- This is not readily expressed in sequence diagrams, much less communication diagrams.
- UML 2.0 introduces **timing diagrams** for this purpose.