The pragmatics of STAIRS

Paper by Ragnhild Kobro Runde, Øystein Haugen and Ketil Stølen

October 5, 2007

INF 5150

Today's topics

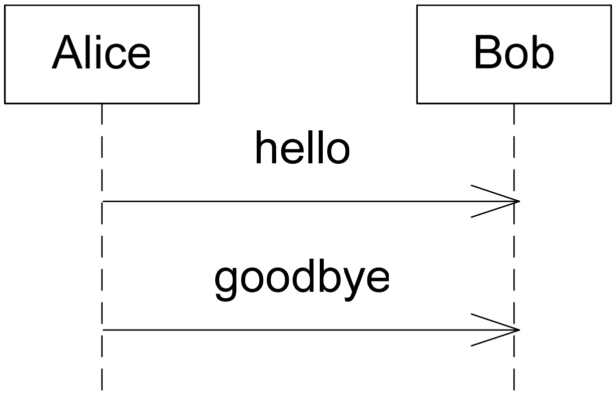
- Explain the practical relevance of STAIRS
- Give guidelines on
  - the use of STAIRS operators
  - refinement
- Illustrated by a running example
- Present some new operators and refinement types
- Some repetition

- The paper can be found on the syllabus/achievement page for INF5150
  - note: updated on Tuesday!

Weak sequencing of trace sets (1)

- $s_1 \succsim s_2$ denotes the set of all traces that may be constructed by selecting one trace t_1 from s_1 and one trace t_2 from s_2 and combining them in such a way that for each lifeline, the events from t_1 comes before the events from t_2.
- Note: if s_1 or s_2 is empty then $s_1 \succsim s_2$ is also empty
- Remember: if the message hello is sent from l_1 to l_2, then the event !hello occurs on l_1 and ?hello occurs on l_2

Weak sequencing of trace sets (2)



Red events occur on Alice, blue events on Bob

s_1 : <!(h,A,B),?(h,A,B)>

$\gtrsim$

s_2 : <!(g,A,B),?(g,A,B)>

=

<!(h,A,B),?(h,A,B),!(g,A,B),?(g,A,B)>

<!(h,A,B),!(g,A,B),?(h,A,B),?(g,A,B)>

$s_1 \gtrsim s_2$ is the set of positive traces for the diagram

Weak sequencing of interaction obligations

- $(p_1,n_1) \succsim (p_2,n_2) \stackrel{\text{def}}{=} (p_1 \succsim p_2\, ,\, (n_1 \succsim p_2) \cup (n_1 \succsim n_2) \cup (p_1 \succsim n_2))$

- Traces composed exclusively by positive traces become positive
- Traces composed with at least one negative trace become negative

Formal semantics of seq

- $[[d_1 \text{ seq } d_2]] \overset{\text{def}}{=} \{o_1 \succsim o_2 \mid o_1 \in [[d_1]] \wedge o_2 \in [[d_2]]\}$

- seq is the implicit composition operator
- o_i is shorthand for (p_i, n_i)
- Note: For better readability we give the binary versions of the operators in this presentation. N-ary versions are used in the paper.

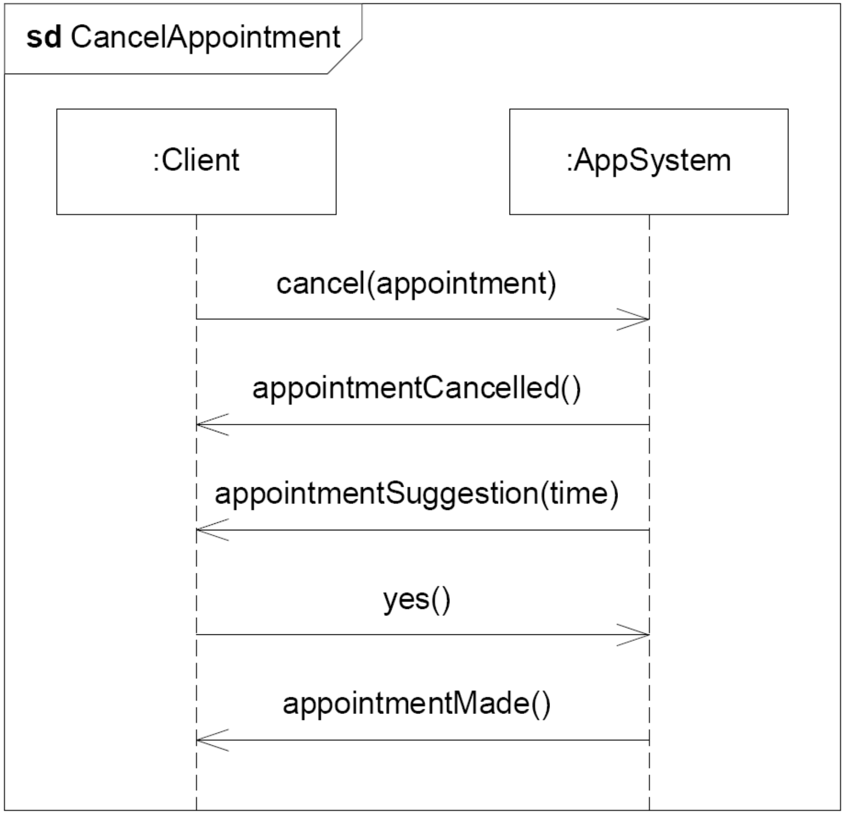
The pragmatics of creating interactions

Example: an appointment system

- A system for booking appointments used by e.g. dentists
- Functionality:
  - MakeAppointment: The client may ask for an appointment
  - CancelAppointment: The client may cancel an appointment
  - Payment: The system may send an invoice message asking the client to pay for the previous or an unused appointment.
- The interactions specifying the system will be developed in a stepwise manner
- Steps will be shown to be valid refinement steps

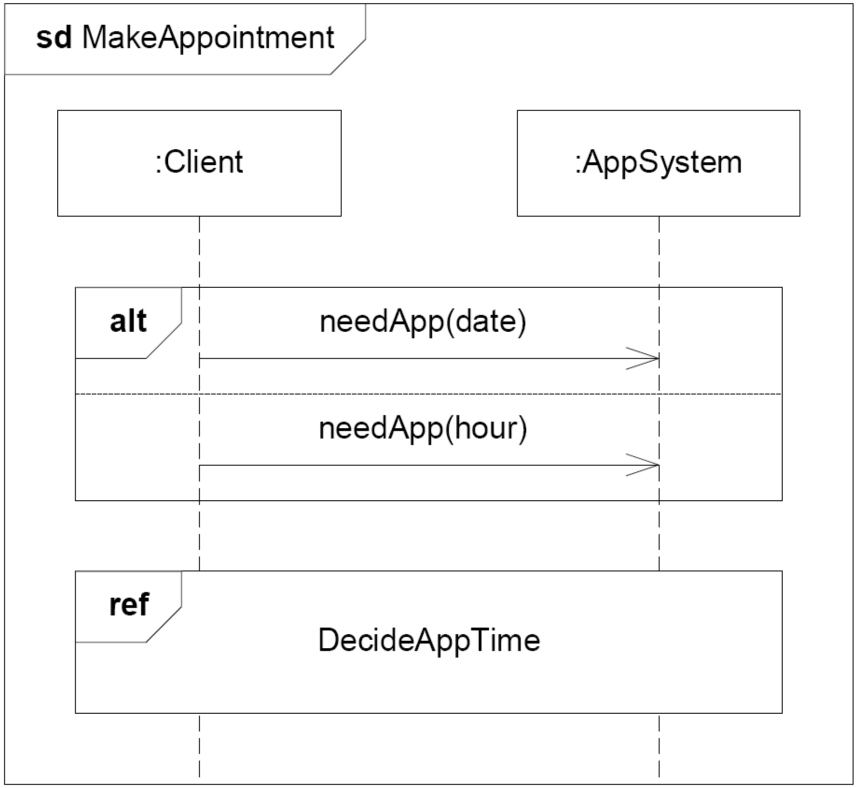
xalt vs alt (1): CancelAppointment

- This specification has two positive traces
- Whether reception of appointmentCancelled() occurs before or after sending of appointmentSuggestion(...) is not important
- Underspecification due to weak sequencing



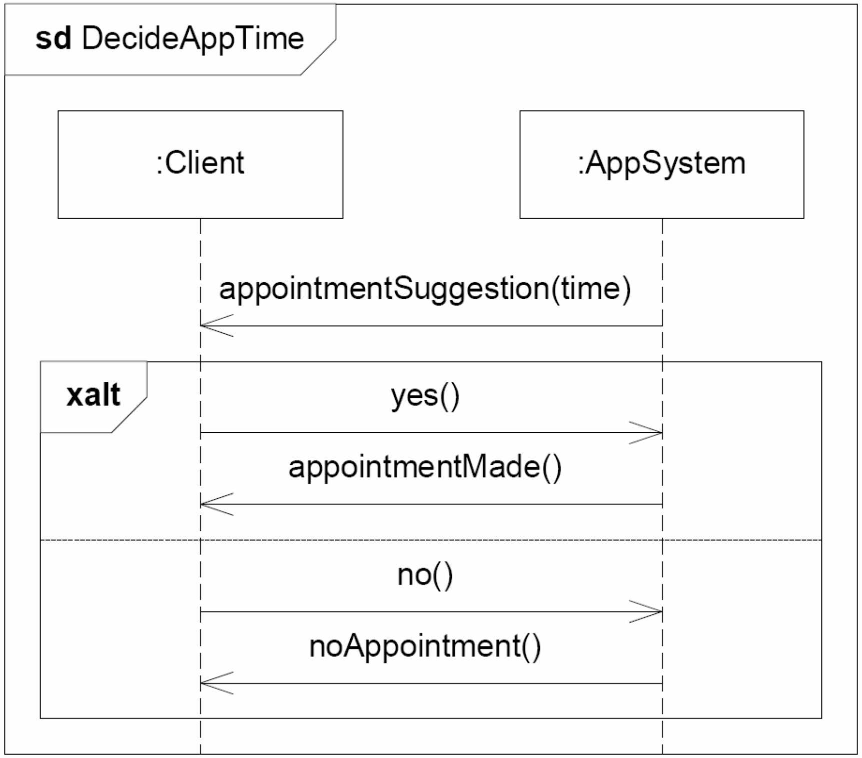
xalt vs alt (2): MakeAppointment

- May ask for either a specific date or a specific hour of the day (e.g. in the lunch break)
- The system is not required to offer both alternatives
- Underspecification expressed by the alt operator



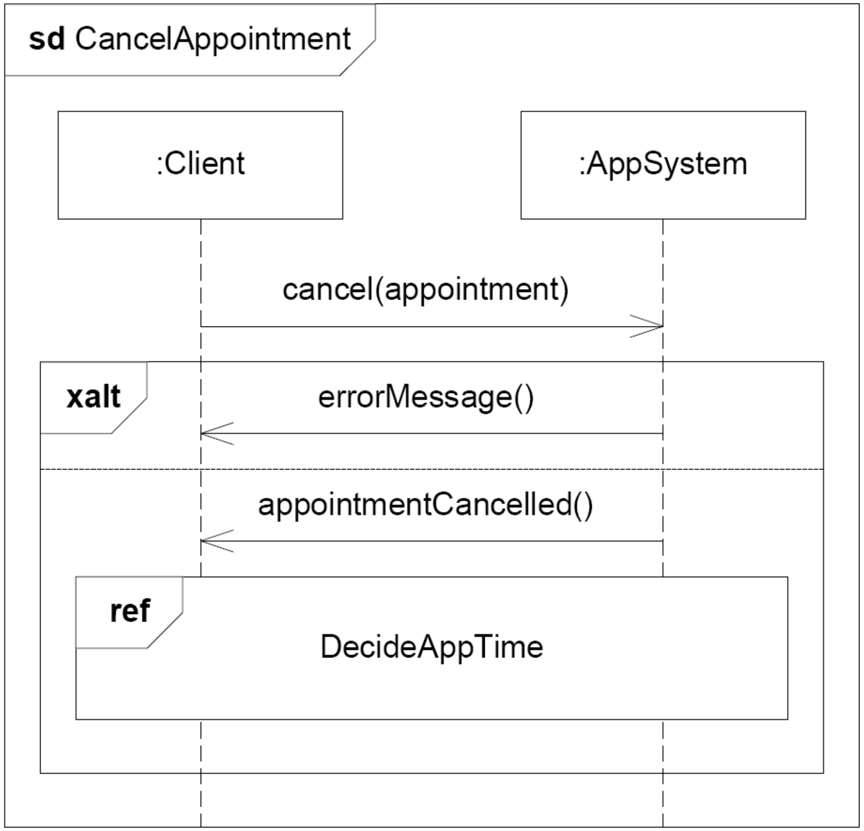
xalt vs alt (3): DecideAppTime

- The system must be able to handle *both* yes() and no() as reply messages from the client
- This is *not* underspecification
- Therefore the alternatives are expressed by the xalt operator



xalt vs alt (4): CancelAppointment

- The condition for choosing errorMessage() or appointmentCancelled() is not shown
- Both alternatives should be possible
- The choice is made by the system



xalt vs alt (5)

- A third use of xalt: to specify inherent nondeterminism
  - for example when specifying a coin toss

- The crucial question when specifying alternatives: <u>Do these alternatives represent similar traces in the sense that implementing only one is sufficient?</u>
  - if yes, use alt
  - otherwise, use xalt

Formal semantics of alt and xalt

- $[[d_1 \text{ alt } d_2]] \stackrel{\text{def}}{=} \{o_1 \uplus o_2 \mid o_1 \in [[d_1]] \wedge o_2 \in [[d_2]]\}$, where
- $(p_1,n_1) \uplus (p_2,n_2) \stackrel{\text{def}}{=} (p_1 \cup p_2,\ n_1 \cup n_2)$

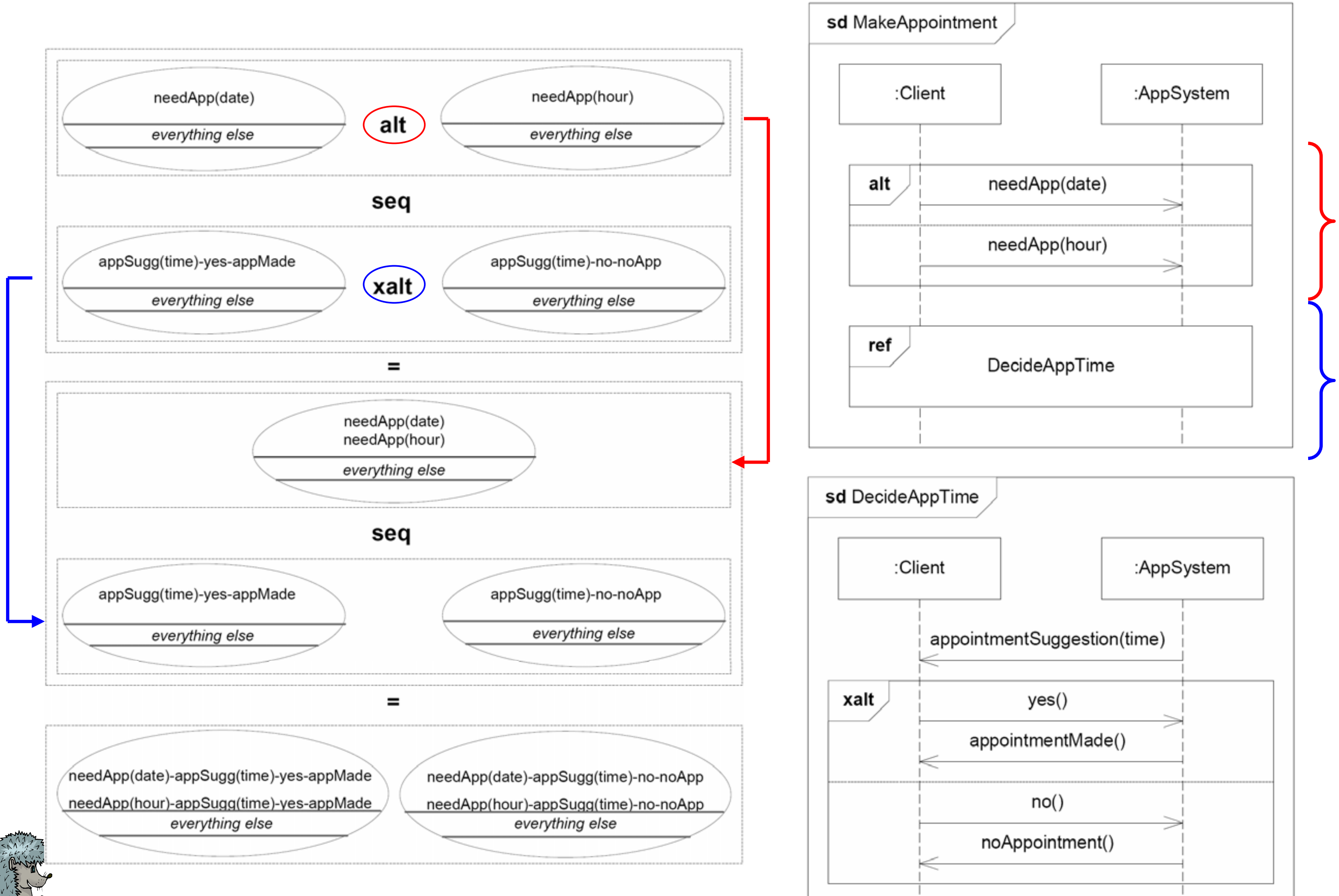
- $[[d_1 \text{ xalt } d_2]] \stackrel{\text{def}}{=} [[d_1]] \cup [[d_2]]$

opt and skip

- $[[\text{opt } d]] \stackrel{\text{def}}{=} [[\text{skip alt } d]]$

- $[[\text{skip}]] \stackrel{\text{def}}{=} \{(\{\langle\rangle\},\emptyset)\}$
  - A single interaction obligation where only the empty trace <> is positive and the set of negative traces is empty

Informal illustration of MakeAppointment

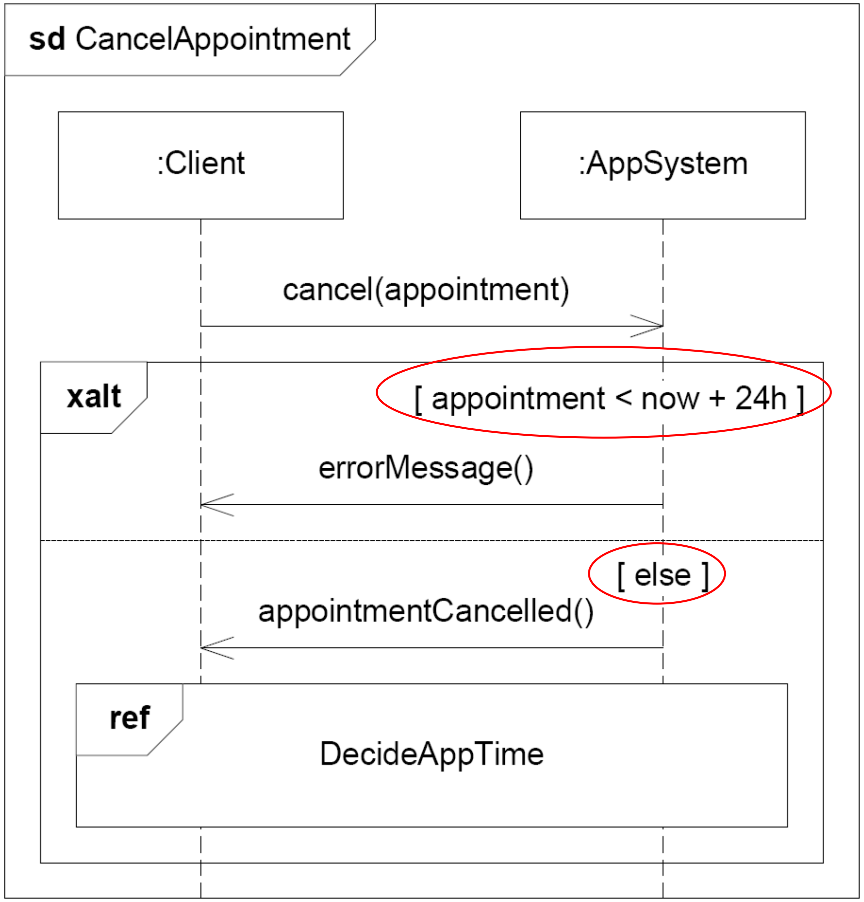


The pragmatics of alt vs xalt

- Use alt to specify alternatives that represent similar traces, i.e. to model
  - underspecification
- Use xalt to specify alternatives that must all be present in an implementation, i.e. to model
  - inherent nondeterminism, as in the specification of a coin toss
  - alternative traces due to different inputs that the system must be able to handle (as in DecideAppTime)
  - alternative traces where the conditions for these being positive are abstracted away (as in CancelAppointment on slide 12)

Guards (1)

- Guards may be used to express conditions for choosing between alternatives
- Here: an error message is sent if the client tries to cancel an appointment less than 24 hours before it is due



Guards (2)

- Semantically, a guard is represented by a special check-event
- The check-event ensures that for each operand to alt/xalt, its traces (including the check-event) become negative if the guard is false
  - otherwise they remain postive or negative as before
- Therefore the guard must be true in all possible situations in which the specified traces are positive
- An alt/xalt operand without a guard can be interpreted as having the guard $\top$ (always true)
- More than one guard may be true at a time

The pragmatics of guards

- Use guards in an alt/xalt construct to constrain the situations in which the different alternatives are positive
- Always make sure that for each alternative, the guard is sufficiently general to capture all possible situations in which the described traces are positive
- In an alt-construct, make sure that the guards are exhaustive. If doing nothing is valid, specify this by using the empty diagram, skip
  - This is in order to avoid confusion with the UML standard

Negative behavior

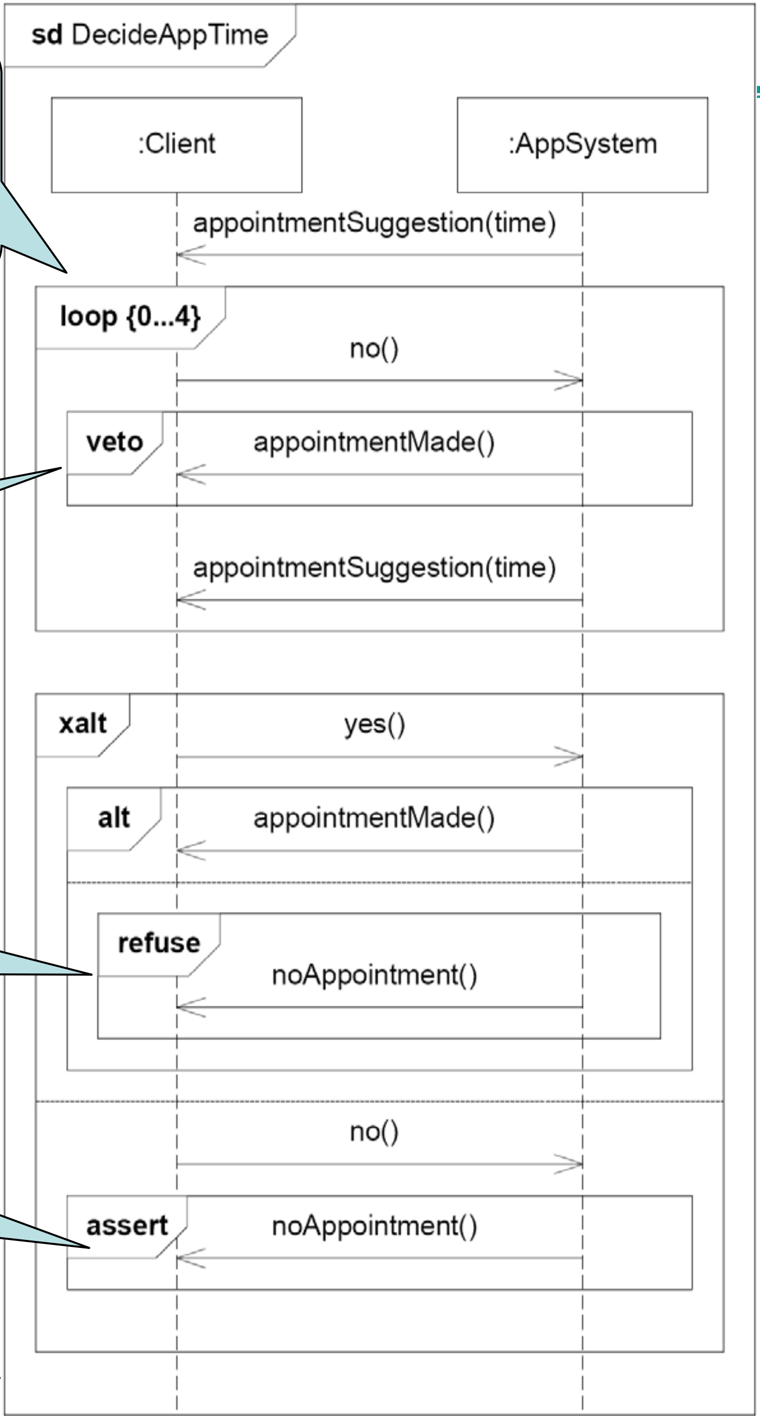
From 0 to 4 iterations (with seq between)

- veto, refuse and assert introduce negative behavior

appointmentMade() may not occur here (veto=neg)

noAppointment() may not occur instead of appointmentMade() here

noAppointment () is the only message that may occur here



refuse

- $[[\text{refuse } d]] \overset{\text{def}}{=} \{(\emptyset, p \cup n) \mid (p, n) \in [[d]]\}$

- All interaction obligations in [[refuse d]] have empty positive sets

- This means that all interaction obligations in [[d_1 seq (refuse d_2)]] have empty positive sets
  - and the same applies to [[(refuse d_1) seq d_2]]

veto

- [[veto d]] $\stackrel{\text{def}}{=}$ [[skip alt (refuse d)]]

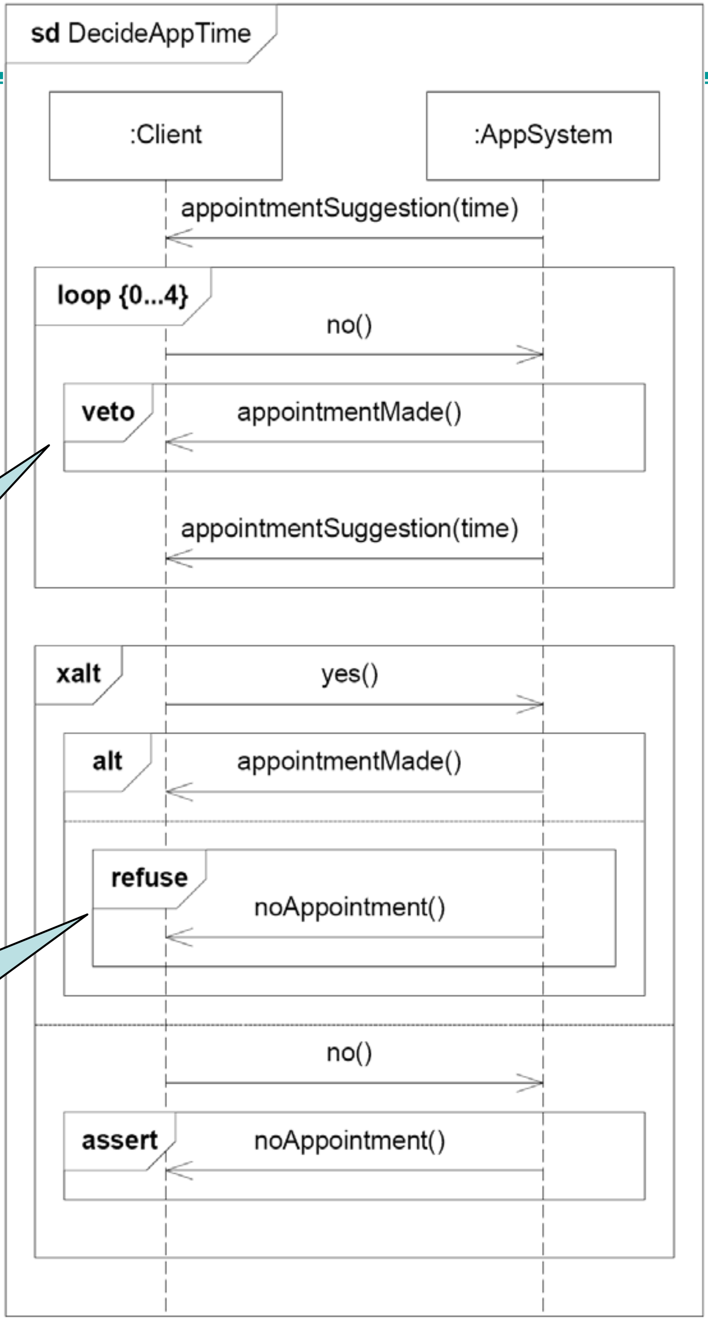
- ... which means that
  $[[\text{veto } d]] = \{(\{\langle\rangle\}, p \cup n) \mid (p \cup n) \in [[d]]\}$
- veto and neg have identical semantics

veto or refuse?

- Should doing nothing be possible in the otherwise negative situation?
  - If yes, use veto
  - If no, use refuse

It is OK to do nothing between no() and appointmentSuggestion(time)

It is not OK to do nothing after yes()

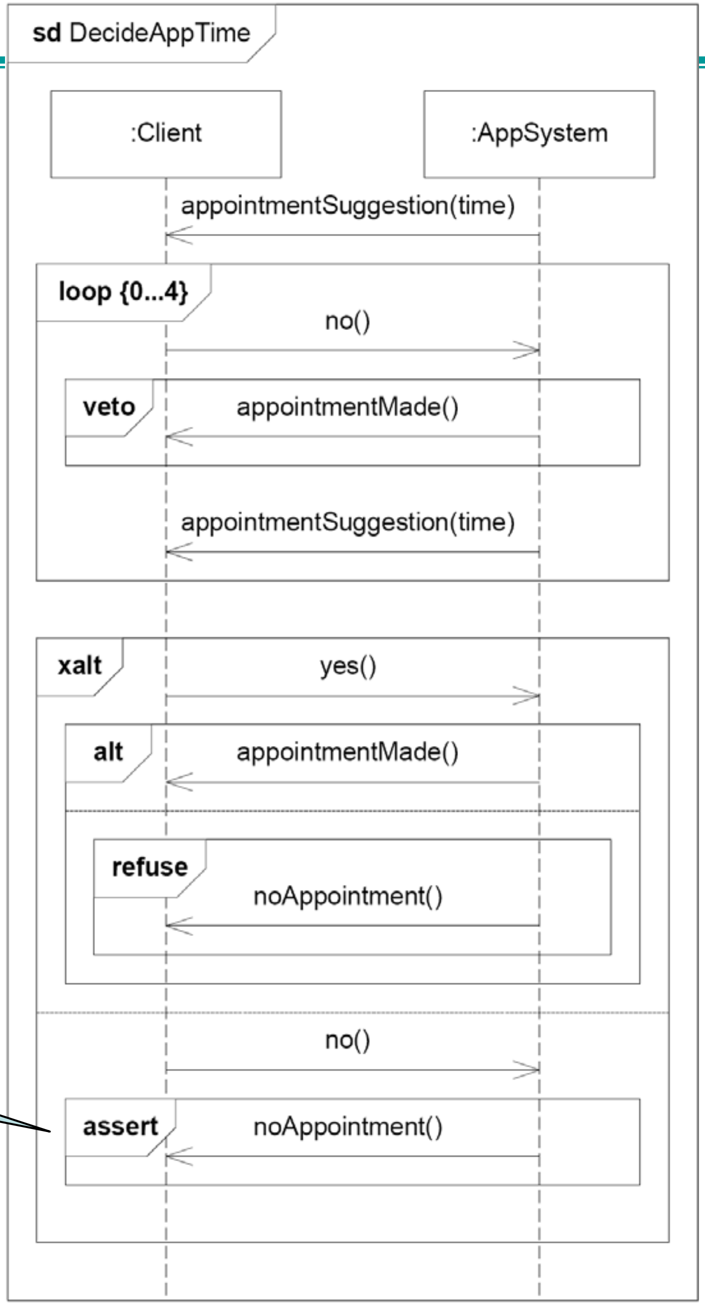


assert (1)

- By using assert, all inconclusive traces are redefined as negative
- This ensures that for each interaction obligation, at least one of its positive traces will be implemented in the final implementation

- $[[\text{assert } d]] \stackrel{\text{def}}{=} \{(p, n \cup (\mathcal{H} \setminus p)) \mid (p,n) \in [[d]]\}$

assert (2)



Sending noAppointment() is the only acceptable response to the no() message at this point

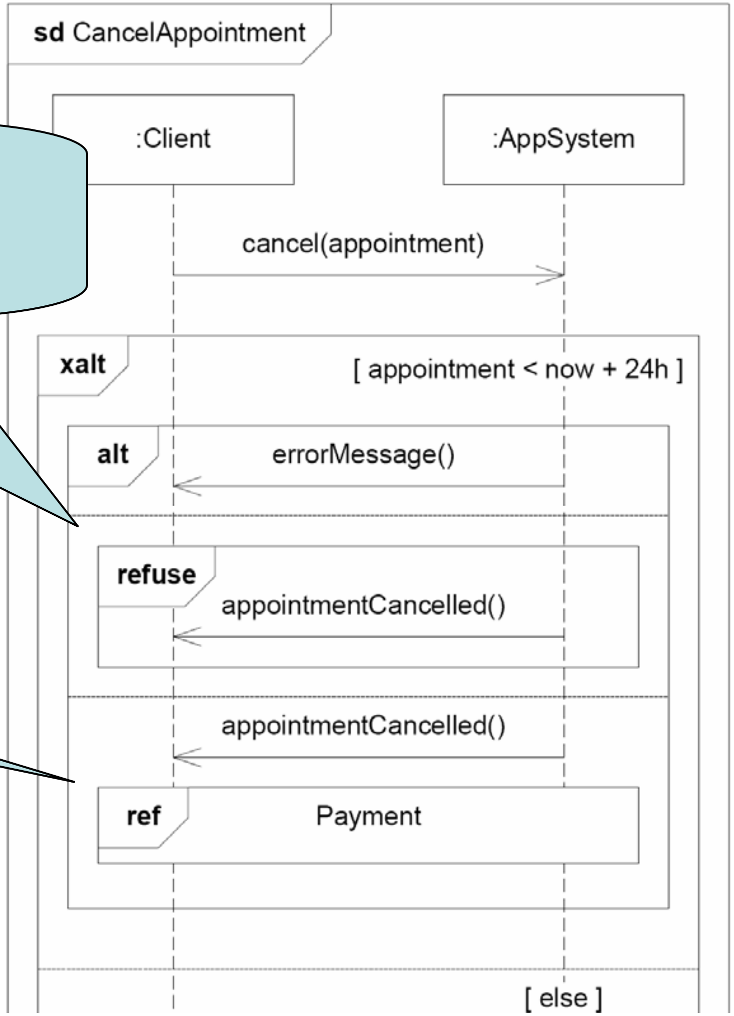
The pragmatics of negation

- To effectively constrain the implementation, the specification should include a reasonable set of negative traces
- Use refuse when specifying that one of the alternatives in an alt-construct represents negative traces
- Use veto when the empty trace (i.e. doing nothing) should be positive, as when specifying a negative message in an otherwise positive scenario
- Use assert on an interaction fragment when all positive traces for that fragment have been described

The use of seq

cancel(appointment) followed by appointmentCancelled() followed by nothing is negative

cancel(appointment) followed by appointmentCancelled() followed by the positive traces of Payment is positive



- A trace is not necessarily negative even if a prefix of it is negative
- The total trace must be considered when categorizing it as positive, negative or inconclusive

The pragmatics of weak sequencing

- Be aware that by weak sequencing
  - a positive sub-trace followed by a positive sub-trace is positive
  - a positive sub-trace followed by a negative sub-trace is negative
  - a negative sub-trace followed by a positive sub-trace is negative
  - a negative sub-trace followed by a negative sub-trace is negative
  - the remaining trace combinations are inconclusive

- Remember the definition:

$$(p_1,n_1) \succsim (p_2,n_2) \stackrel{\text{def}}{=} (p_1 \succsim p_2 , (n_1 \succsim p_2) \cup (n_1 \succsim n_2) \cup (p_1 \succsim n_2))$$

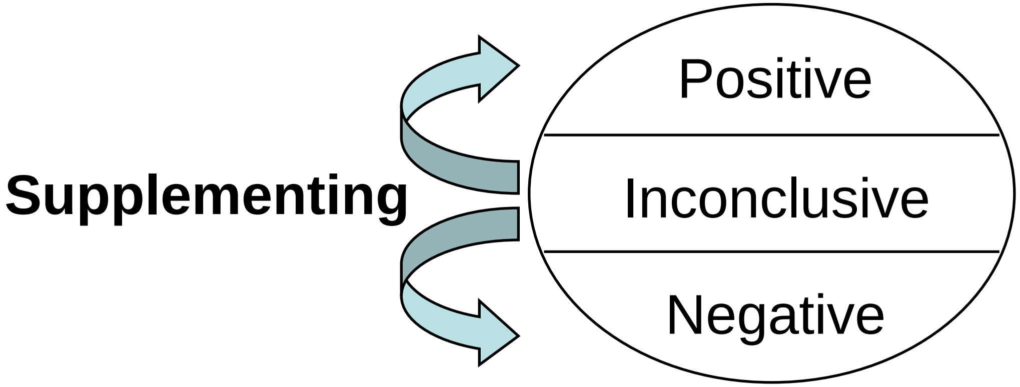
The pragmatics of refining interactions

The use of supplementing

- Inconclusive trace are recategorized as either positive or negative (for an interaction obligation)
- New situations are considered
  - adding fault tolerance
  - new user requirements
  - ...
- Typically used in early phases

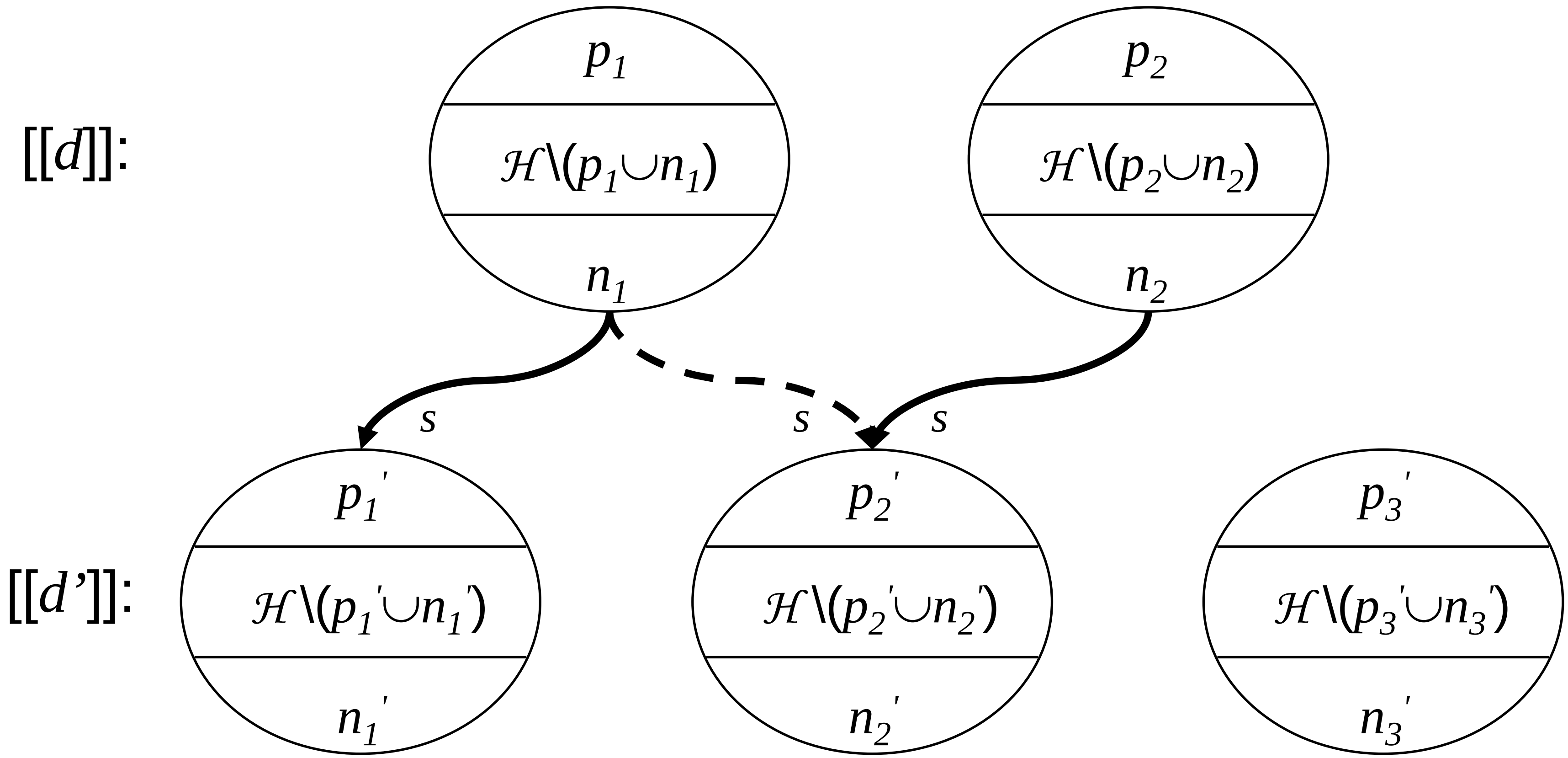
Supplementing of interaction obligations

- $(p,n) \rightsquigarrow_{s} (p',n') \stackrel{\text{def}}{=} p \subseteq p' \land n \subseteq n'$

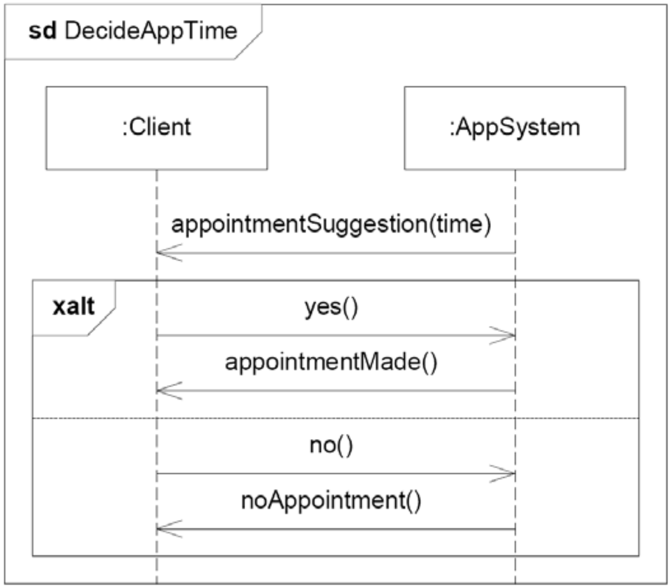


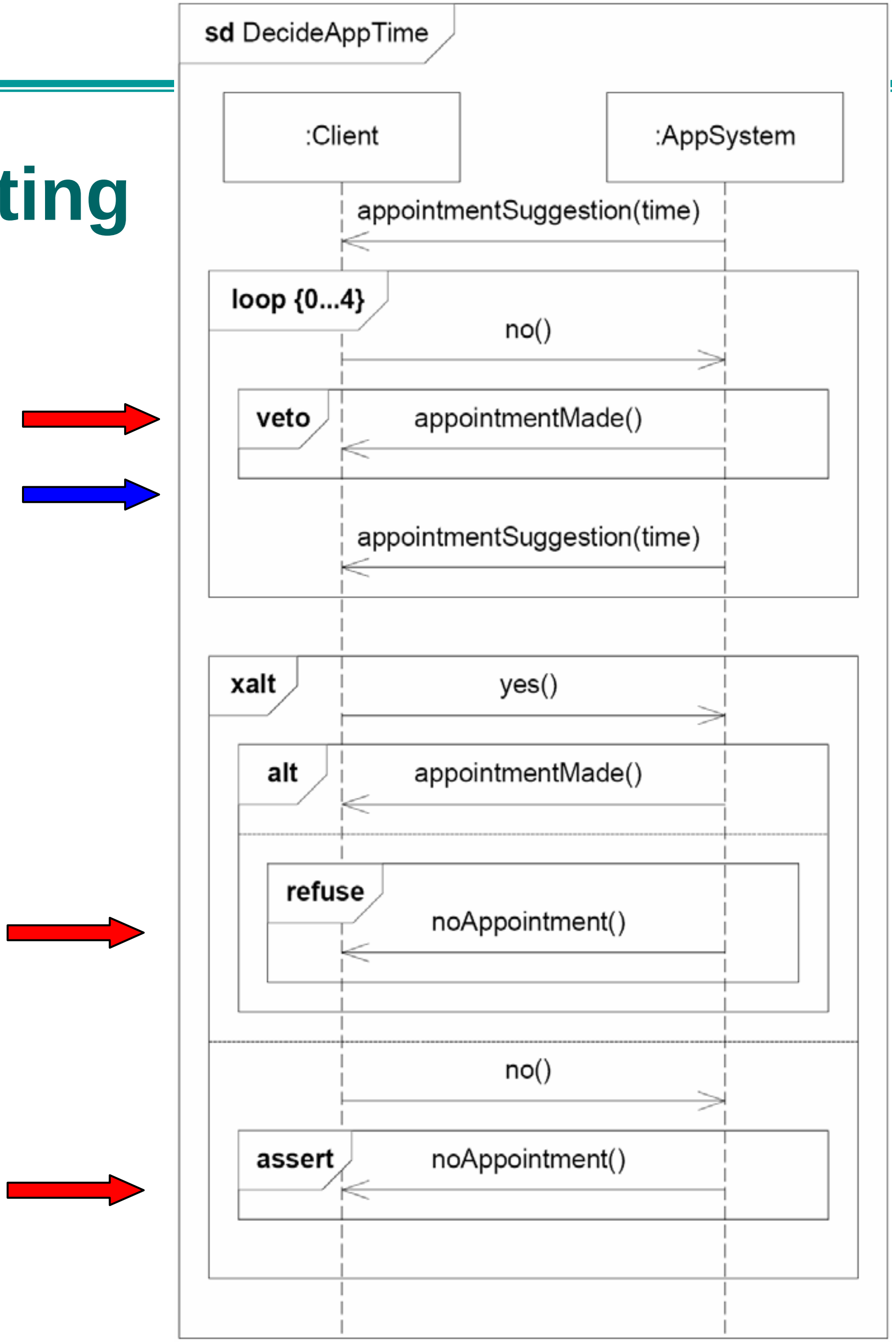
Supplementing of specifications

- $d \rightsquigarrow_s d' \stackrel{\text{def}}{=} \forall o \in [[d]] : \exists o' \in [[d']] : o \rightsquigarrow_s o'$



Example of supplementing



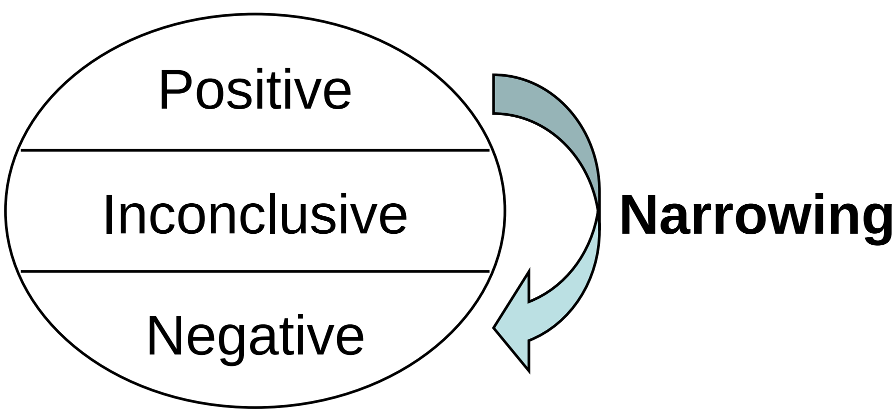


The pragmatics of supplementing

- Use supplementing to add positive or negative traces to the specification
- When supplementing, all of the original positive traces must remain positive, and all of the original negative traces must remain negative
- Do not use supplementing on the operand of an assert
  - no traces are inconclusive in the operand

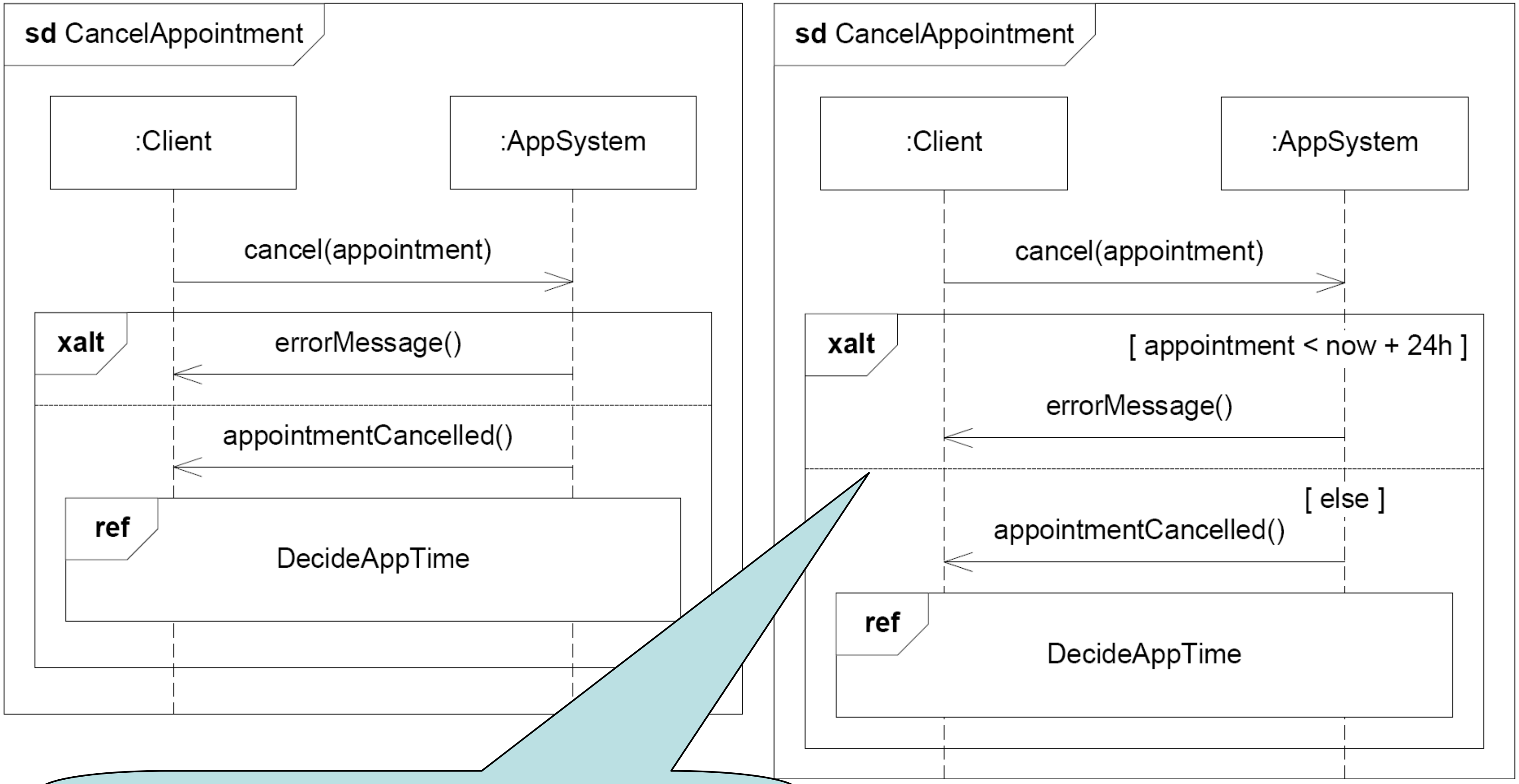
Narrowing

- Reduce underspecification by redefining positive traces as negative
- For example adding guards, or replacing a guard with a stronger one
  - traces where the guard is false become negative
- $(p,n) \rightsquigarrow_n (p',n') \stackrel{\text{def}}{=} p' \subseteq p \wedge n' = n \cup (p \setminus p')$
- $d \rightsquigarrow_n d' \stackrel{\text{def}}{=} \forall o \in [[d]] : \exists o' \in [[d']] : o \rightsquigarrow_n o'$



INF 5150

Example of narrowing



For each operand, traces where the guard is false become negative

The pragmatics of narrowing

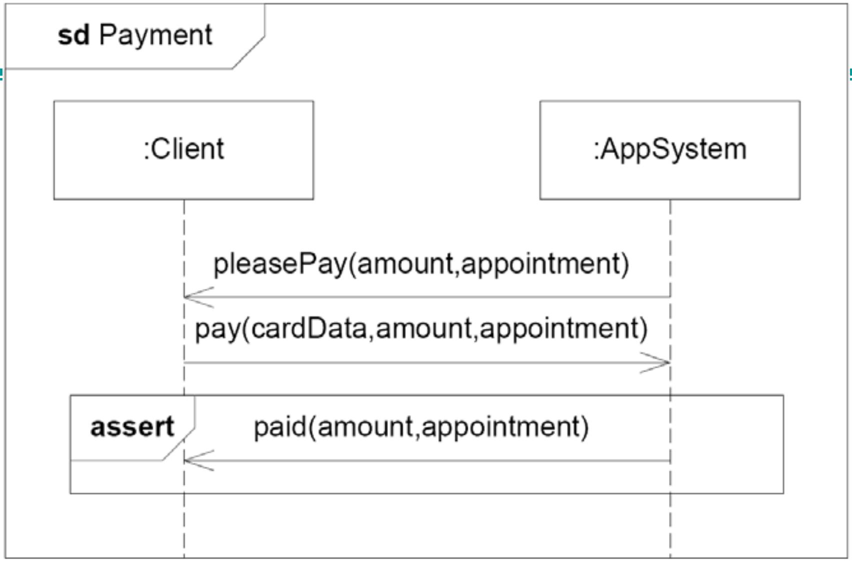
- Use narrowing to remove underspecification by redefining postive traces as negative
- In cases of narrowing, all of the original negative traces must remain negative
- Guards may be added to an alt-construct as a legal narrowing step
- Guards may be added to an xalt-construct as a legal narrowing step
- Guards may be narrowed, i.e. the refined condition must imply the original one

The use of detailing

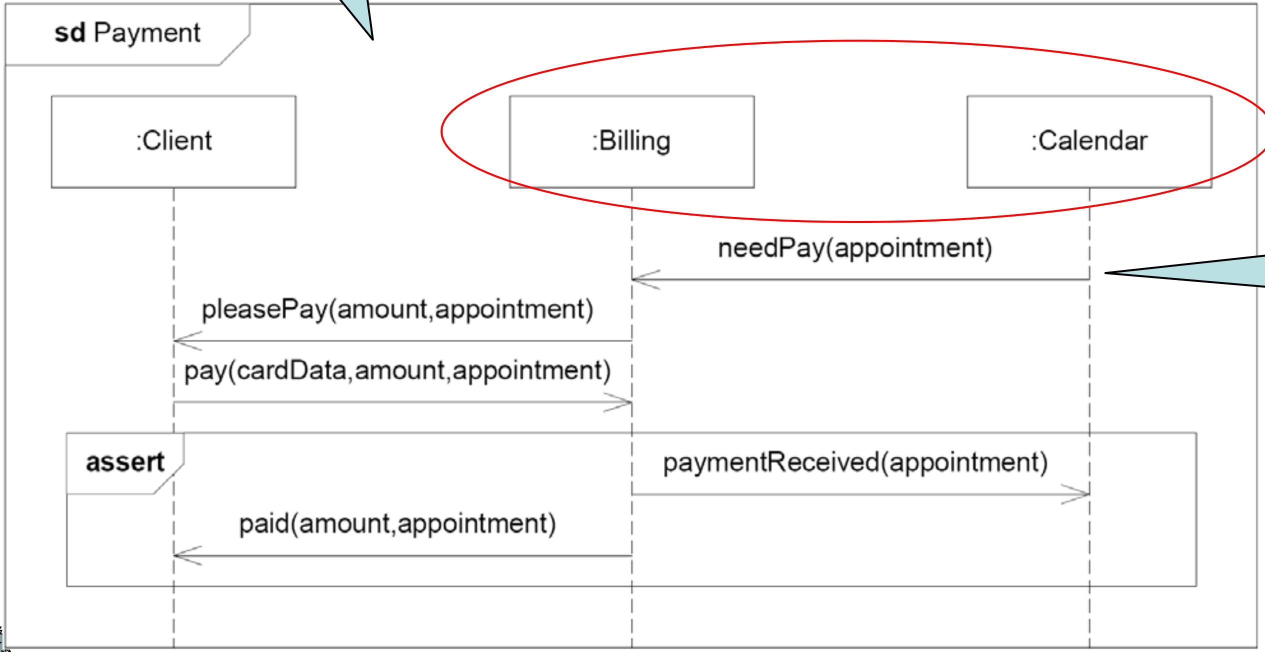
- Reducing the level of abstraction by structural decomposition
  - One or more lifelines are decomposed
- The positive and the negative traces are the same, except that
  - internal communication is hidden at the abstract level
  - events occuring on a composed lifeline at the abstract level occur instead on one of the component lifelines

UNIVERSITY OF OSLO

Example of detailing



Note that a UML principle has been broken here.



Components of AppSystem

Internal communication

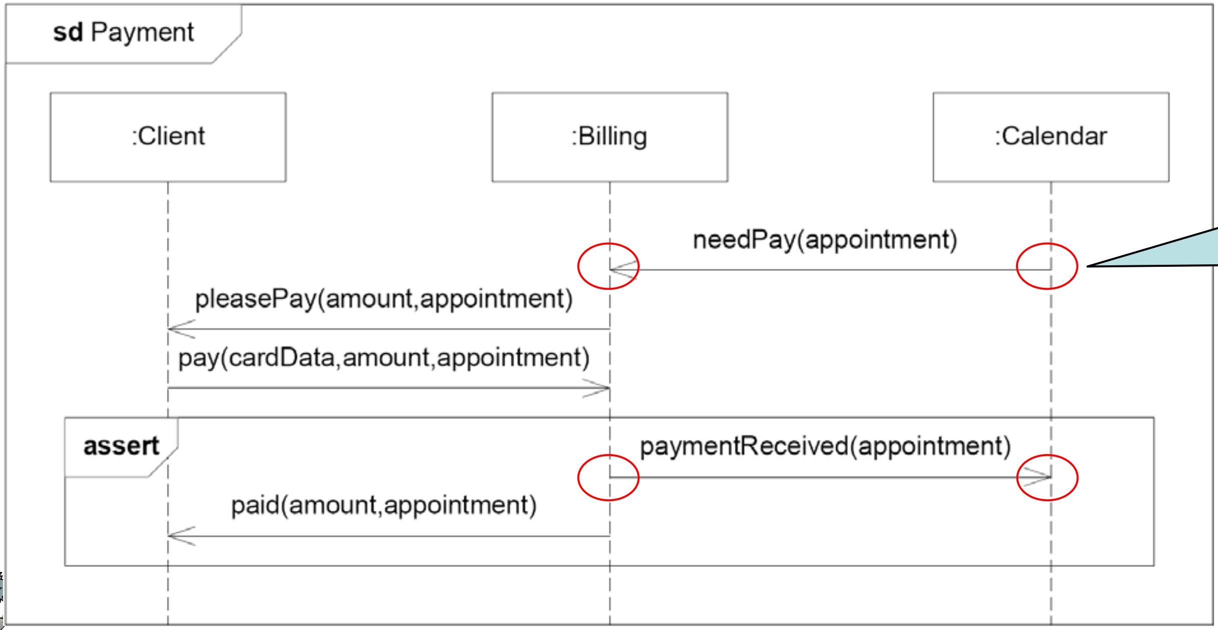
INF 5150

Detailing

- L is a mapping that defines the translation from concrete to abstract lifelines
  - L={Client↦Client, Billing↦AppSystem, Calendar↦AppSystem}
  - This implies that Billing and Calendar are components of AppSystem
- $subst(t,L)$ is a function that substitutes lifelines in the trace t according to L
- E is a set of abstract events
  - Necessary to allow messages that an abstract lifeline sends to itself to be visible in the abstract diagram
- $abstr(s,L,E)$ is an abstraction function that transforms a set of concrete traces s into a set of abstract traces
  - by removing all internal events (w.r.t. L) that are not in E

Formal definition of detailing

- $(p,n) \rightsquigarrow_c^{L,E} (p',n') \stackrel{\text{def}}{=} p=\text{abstr}(p',L,E) \wedge n=\text{abstr}(n',L,E)$
- $d \rightsquigarrow_c^{L,E} d' \stackrel{\text{def}}{=} \forall o \in [[d]] : \exists o' \in [[d']] : o \rightsquigarrow_c^{L,E} o'$



Internal events not visible at the abstract level

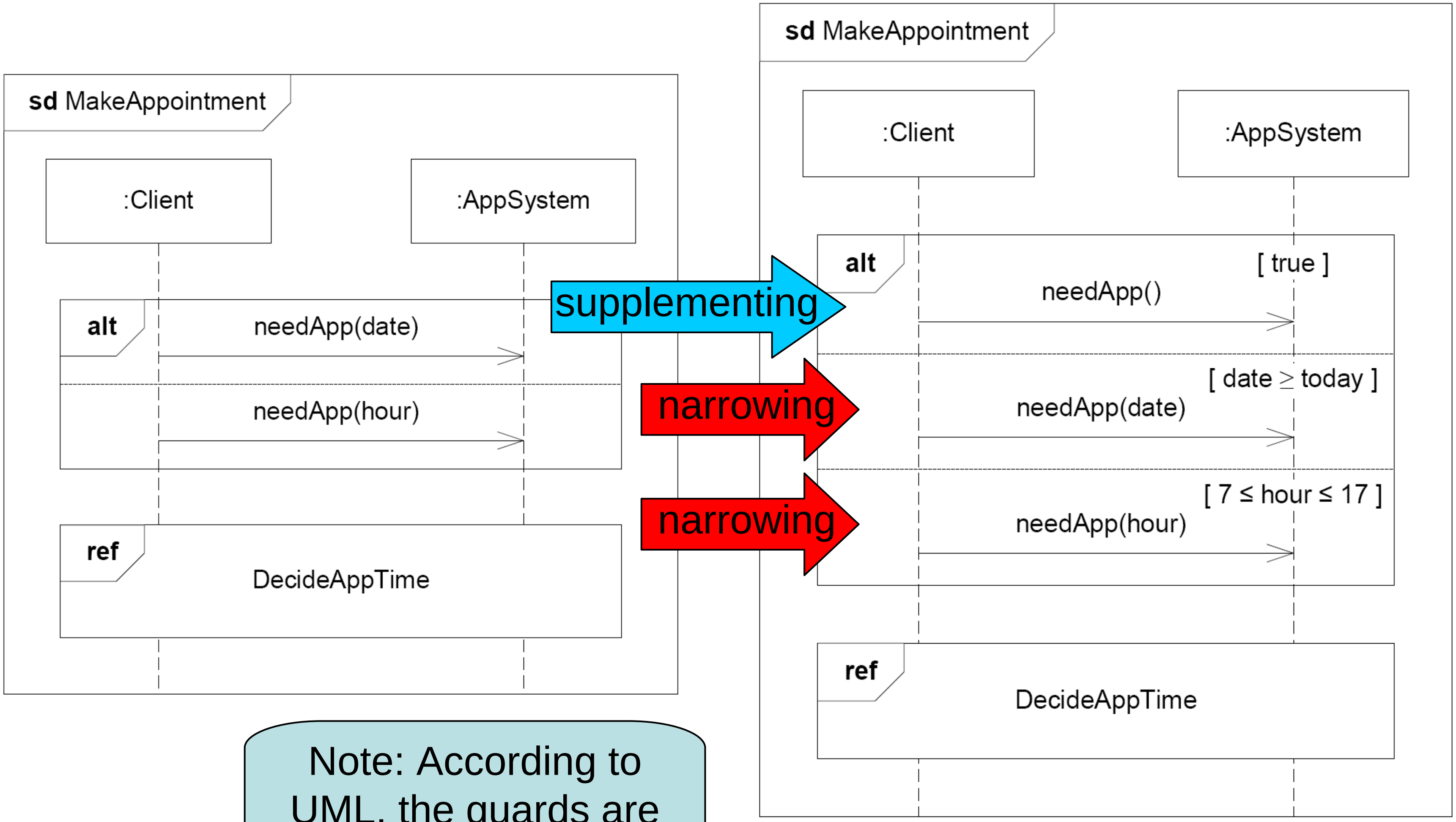
The pragmatics of detailing

- Use detailing to increase the level of granularity of the specification by decomposing lifelines
- When detailing, document the decomposition by creating a mapping L from the concrete to the abstract lifelines
- When detailing, make sure that the refined traces are equal to the original ones when abstracting away internal communication and taking the lifeline mapping into account

The use of general refinement

- A combination of supplementing, narrowing and detailing
  - (not necessarily all three)
- Allows all positive traces to become negative, while previosly inconclusive traces become positive
- To ensure that a trace *must* be present in the final implementation we need an interaction obligation where all other traces are negative

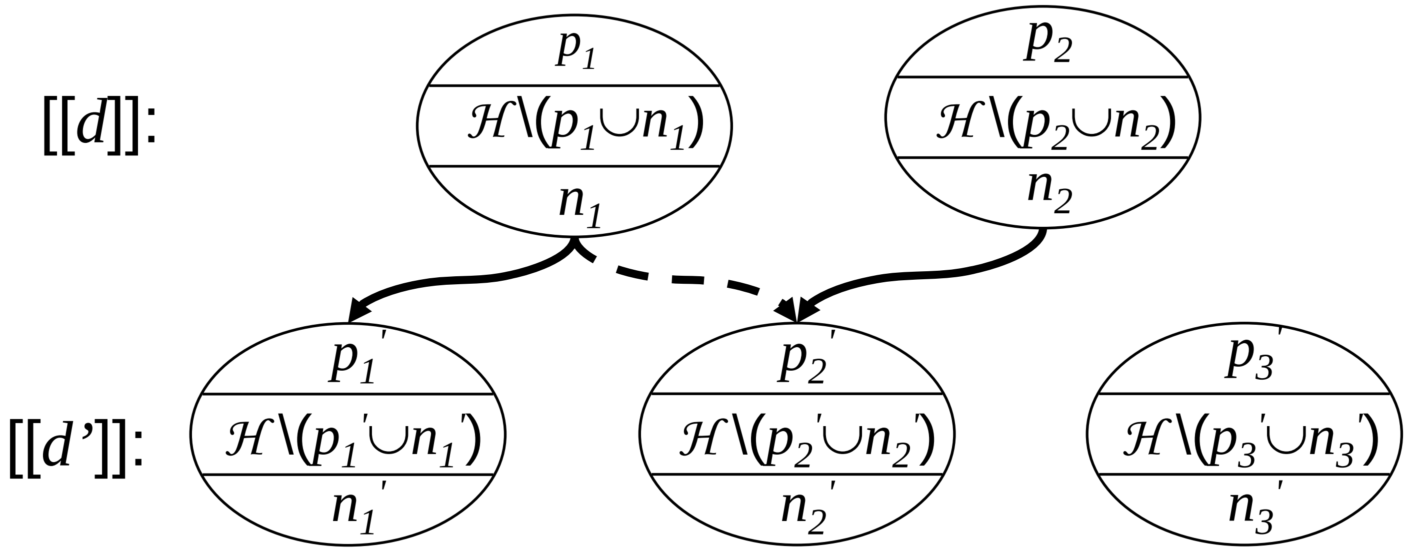
Example of general refinement



Note: According to UML, the guards are on the wrong lineline

General refinement (of sets of interaction obligations)

- $d \rightsquigarrow d' \overset{\text{def}}{=} \forall o \in [[d]] : \exists o' \in [[d']] : o \rightsquigarrow o'$
- *d'* is a general refinement of *d* if
  - for every interaction obligation *o* in [[*d*]] there is at least one interaction obligation *o'* in [[*d'*]] such that *o'* is a general refinement of *o*
- New interaction obligations may also be added
  - that do not refine any obligation at the abstract level



The pragmatics of general refinement

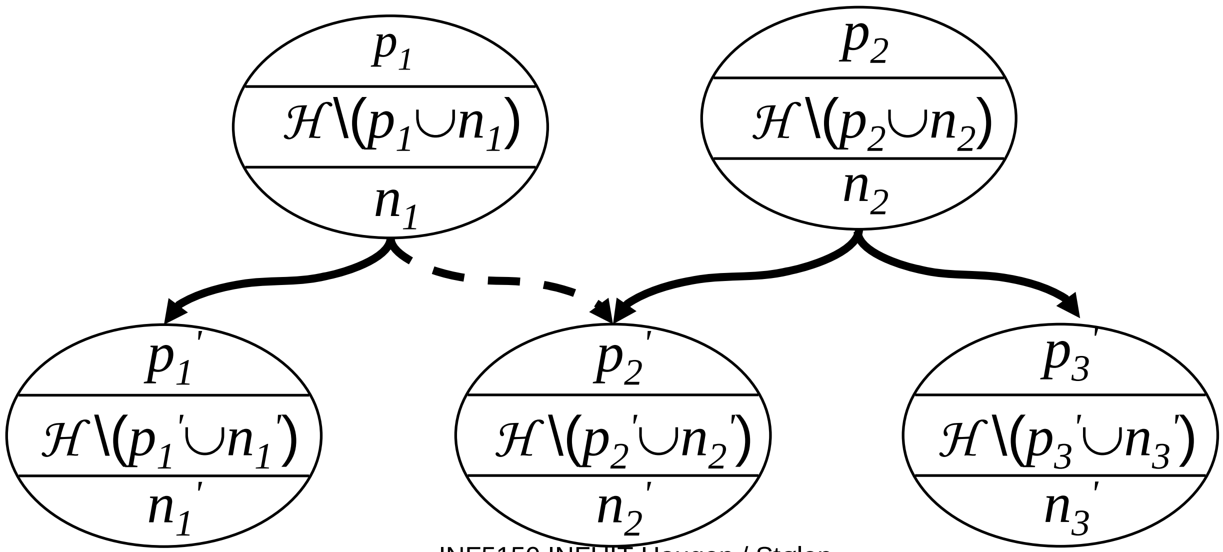
- Use general refinement to perform a combination of supplementing, narrowing and detailing in a single step
- To define that a particular trace *must* be present in an implementation use xalt and assert to characterize an obligation with this trace as the only positive one and all other traces as negative

Limited refinement

- Limits the possibility of adding new interaction obligations
- Typically used at a later stage
- *d'* is a limited refinement of *d* if
  - *d'* is a general refinement of *d*, and
  - every interaction obligation in [[*d'*]] is a general refinement of at least one interaction obligation in [[*d*]]

[[*d*]]:

[[*d'*]]:



The pragmatics of limited refinement

- Use assert and switch to limited refinement in order to avoid fundamentally new traces being added to the specification
- To specify globally negative traces, define these as negative in all operands of xalt, and switch to limited refinement

Compositionality

- A refinement operator $\leadsto$ is compositional if it is
  - reflexive: $d \leadsto d$
  - transitive: $d \leadsto d' \wedge d' \leadsto d'' \Rightarrow d \leadsto d''$
  - the operators refuse, veto, alt, xalt and seq are monotonic w.r.t. $\leadsto$:
    - $d \leadsto d' \Rightarrow$ refuse $d \leadsto$ refuse d'
    - $d \leadsto d' \Rightarrow$ veto $d \leadsto$ veto d'
    - $d_1 \leadsto d_1' \wedge d_2 \leadsto d_2' \Rightarrow d_1$ alt $d_2 \leadsto d_1'$ alt d_2'
    - $d_1 \leadsto d_1' \wedge d_2 \leadsto d_2' \Rightarrow d_1$ xalt $d_2 \leadsto d_1'$ xalt d_2'
    - $d_1 \leadsto d_1' \wedge d_2 \leadsto d_2' \Rightarrow d_1$ seq $d_2 \leadsto d_1'$ seq d_2'
- Transitivity allows stepwise development
- Monotonicity allow different parts of the specification to be refined separately
- Supplementing, narrowing, detailing, general refinement and limited refinement are all compositional ☺