



Refinement III

Ketil Stølen





Topics

- Weak sequencing
- Negative behaviour (refuse, veto, assert)
- Refinement (pragmatics of refining interactions)

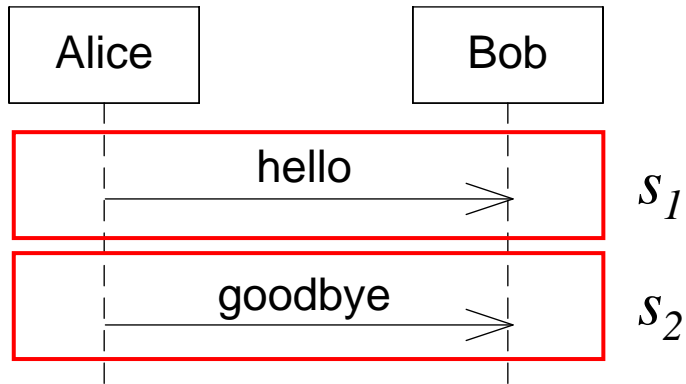


Weak sequencing

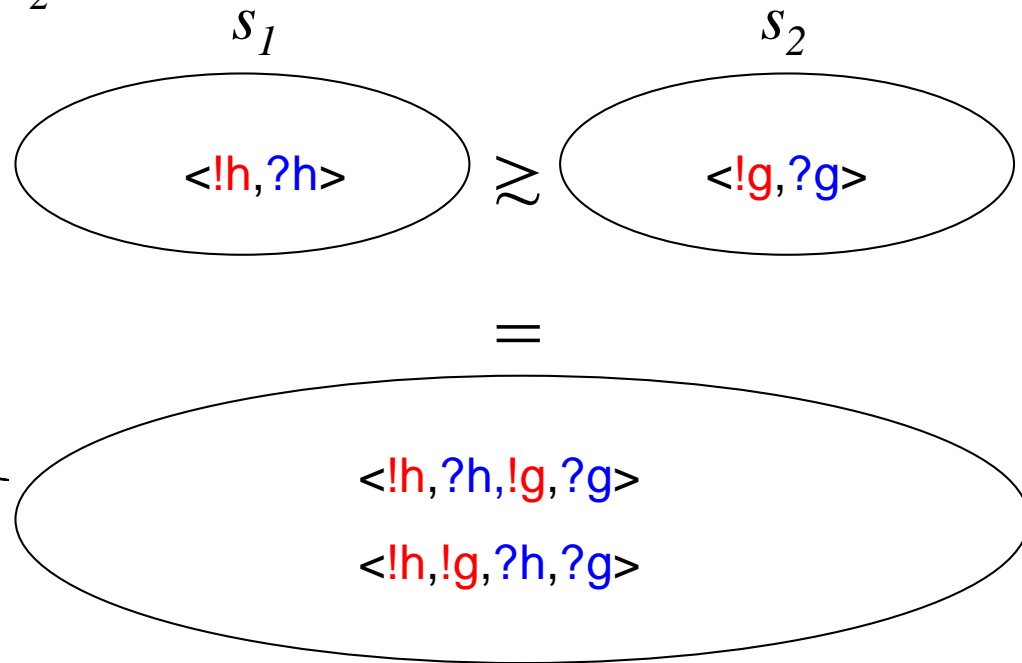
- Combine interaction fragments by seq
- Definition of weak sequencing of trace sets:
- $s_1 \succcurlyeq s_2$ denotes the set of all traces that may be constructed by selecting one trace t_1 from s_1 and one trace t_2 from s_2 and combining them in such a way that for each lifeline, the events from t_1 comes before the events from t_2 .
- Note: if s_1 or s_2 is empty then $s_1 \succcurlyeq s_2$ 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



Red events occur on Alice,
blue events on Bob



$S_1 \approx S_2$ 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

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

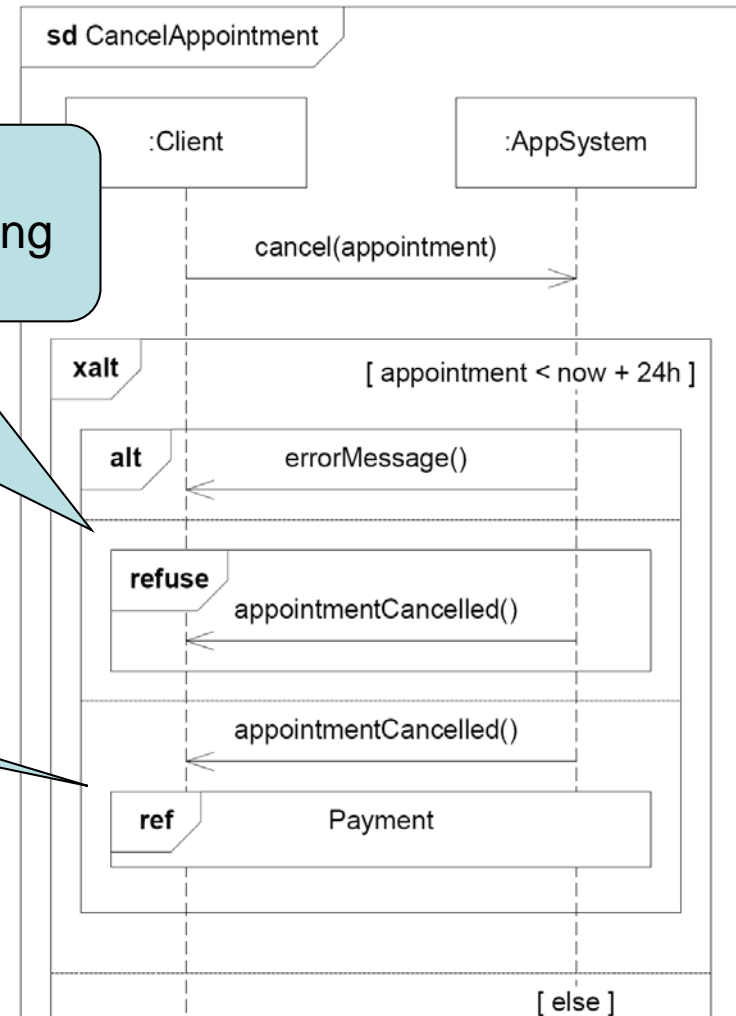


Note

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:

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



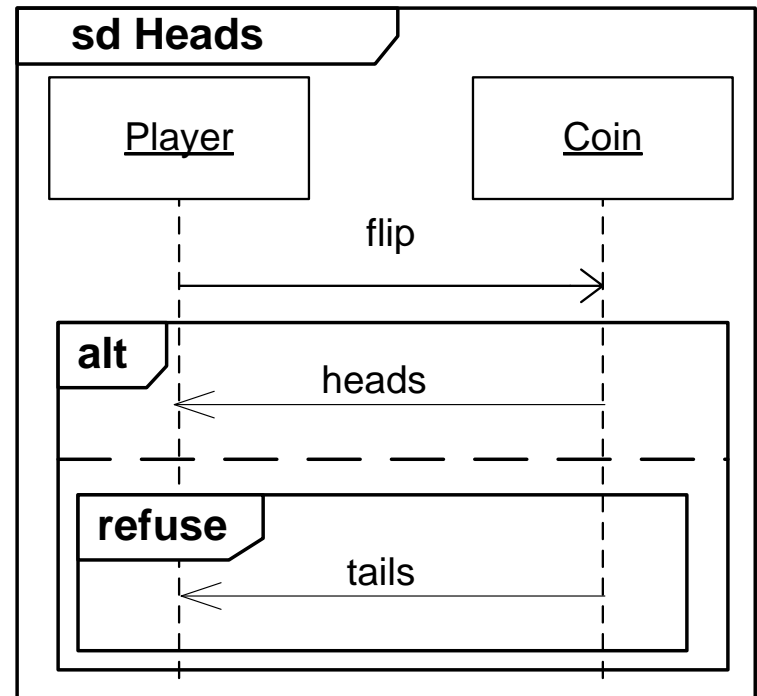
opt and skip

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



Specifying negative behaviour: refuse

- $[[\text{refuse } d]] \stackrel{\text{def}}{=} \{(\{\}, p \cup n) \mid (p, n) \in [[d]]\}$
- All interaction obligations in $[[\text{refuse } d]]$ have empty positive sets
- This means that all interaction obligations in $[[d_1 \text{ seq } (\text{refuse } d_2)]]$ have empty positive sets
 - and the same applies to $[[(\text{refuse } d_1) \text{ seq } d_2]]$

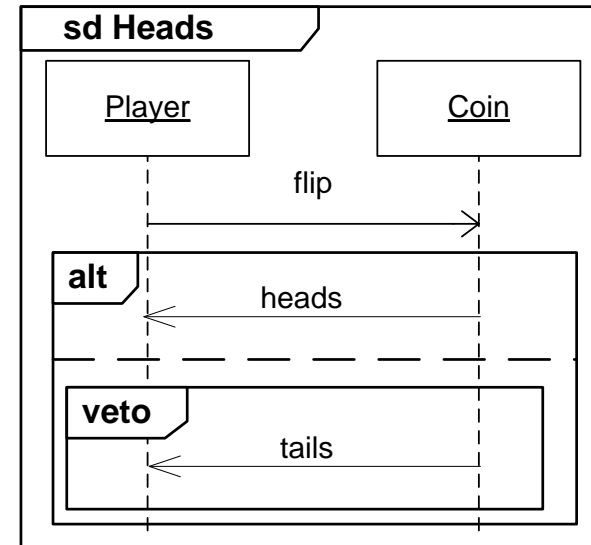


- $[[\text{Heads}]] = \{(\{<!f, ?f, !h, ?h>, <!f, ?f, !t, ?t>\})\}$



Specifying negative behaviour: veto

- $[[\text{veto } d]] \stackrel{\text{def}}{=} [[\text{skip alt (refuse } d)]]$
- ... which means that
 $[[\text{veto } d]] = \{(\{\langle \rangle\}, p \cup n) \mid (p \cup n) \in [[d]]\}$

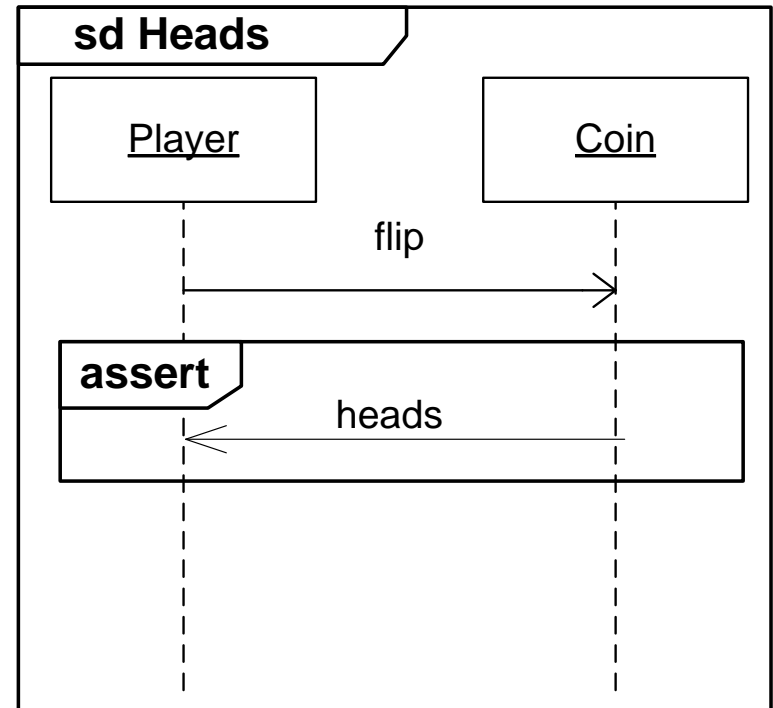


- $[[\text{Heads}]] = \{(\{\langle !f, ?f, !h, ?h \rangle, \langle !f, ?f \rangle\}, \{\langle !f, ?f, !t, ?t \rangle\})\}$



Specifying negative behaviour : assert

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



- $[[\text{Heads}]] = \{(\langle !f, ?f, !h, ?h \rangle, n)\}$
- n = all traces where the first event on the lifeline of Player is !f and the first event on the lifeline of Coin is ?f except the trace $\langle !f, ?f, !h, ?h \rangle$



Negative behaviour

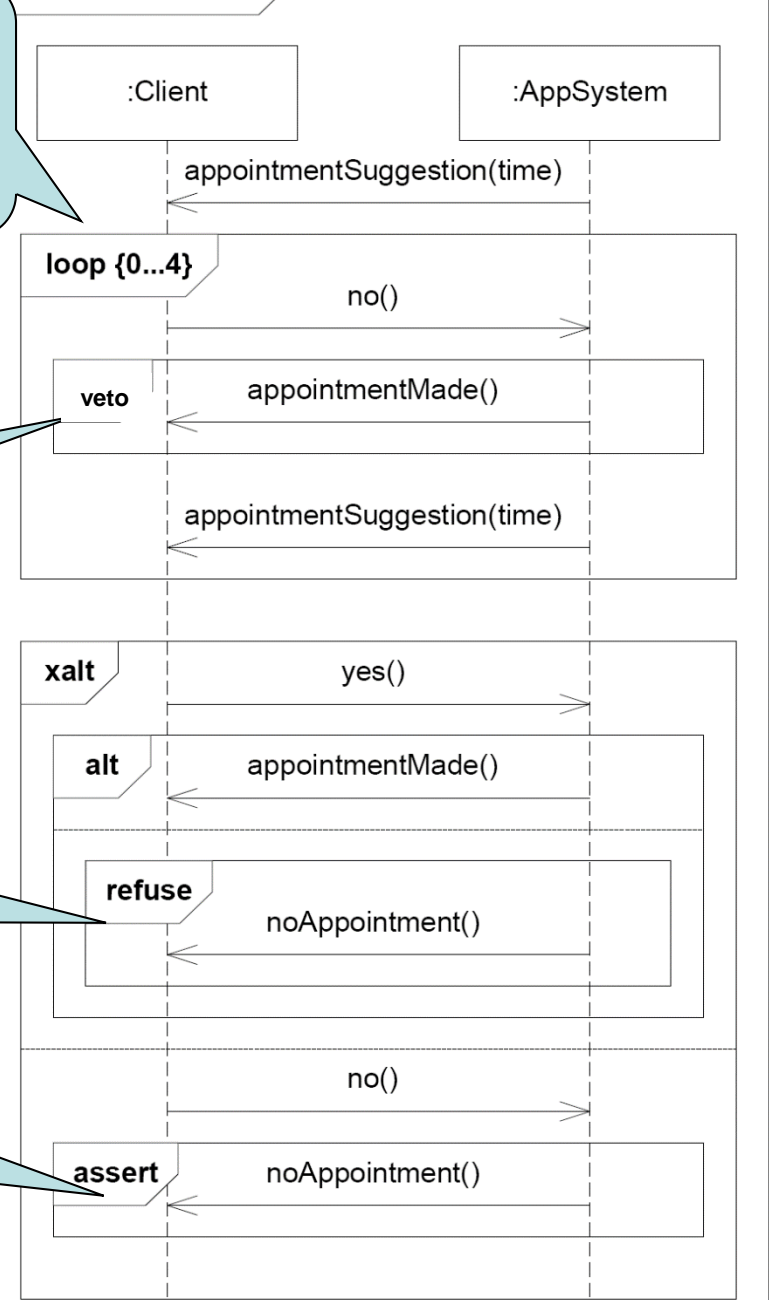
From 0 to 4 iterations (with seq between)

appointmentMade() may not occur here

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

noAppointment () is the only message that may occur here

sd DecideAppTime

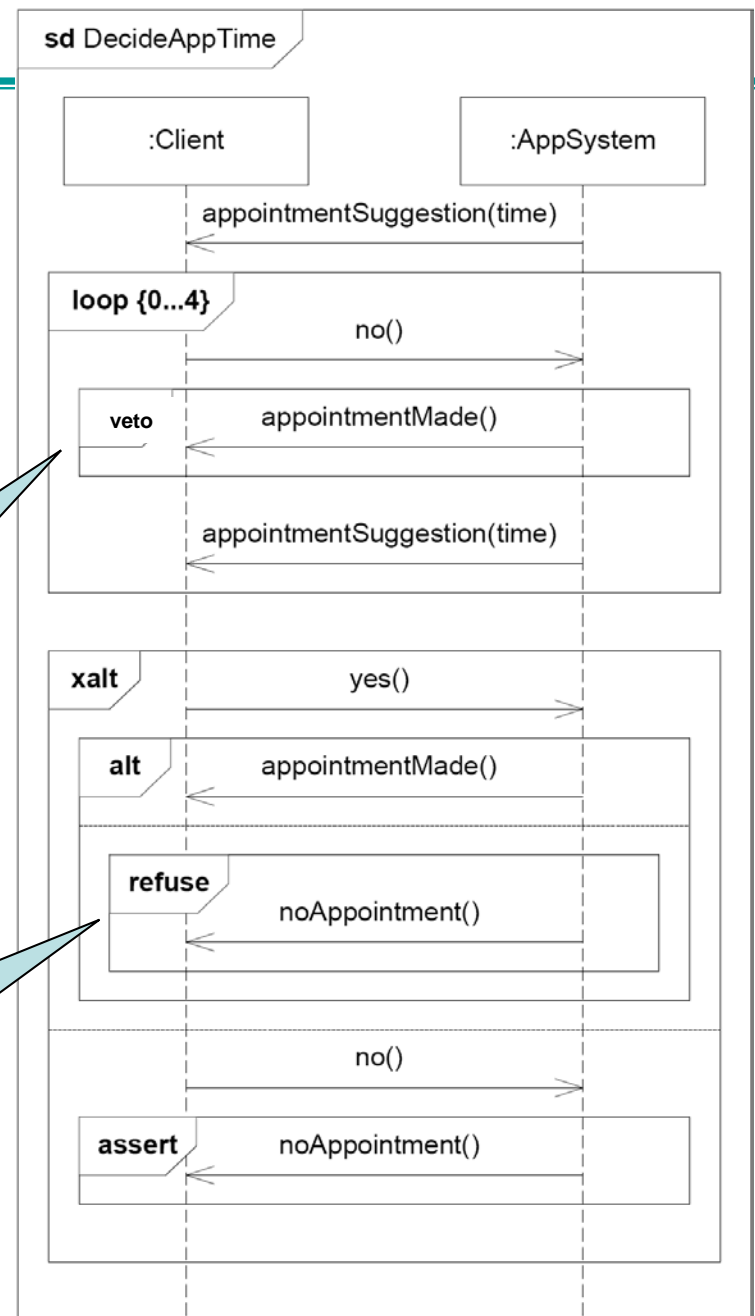


veto or refuse?

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

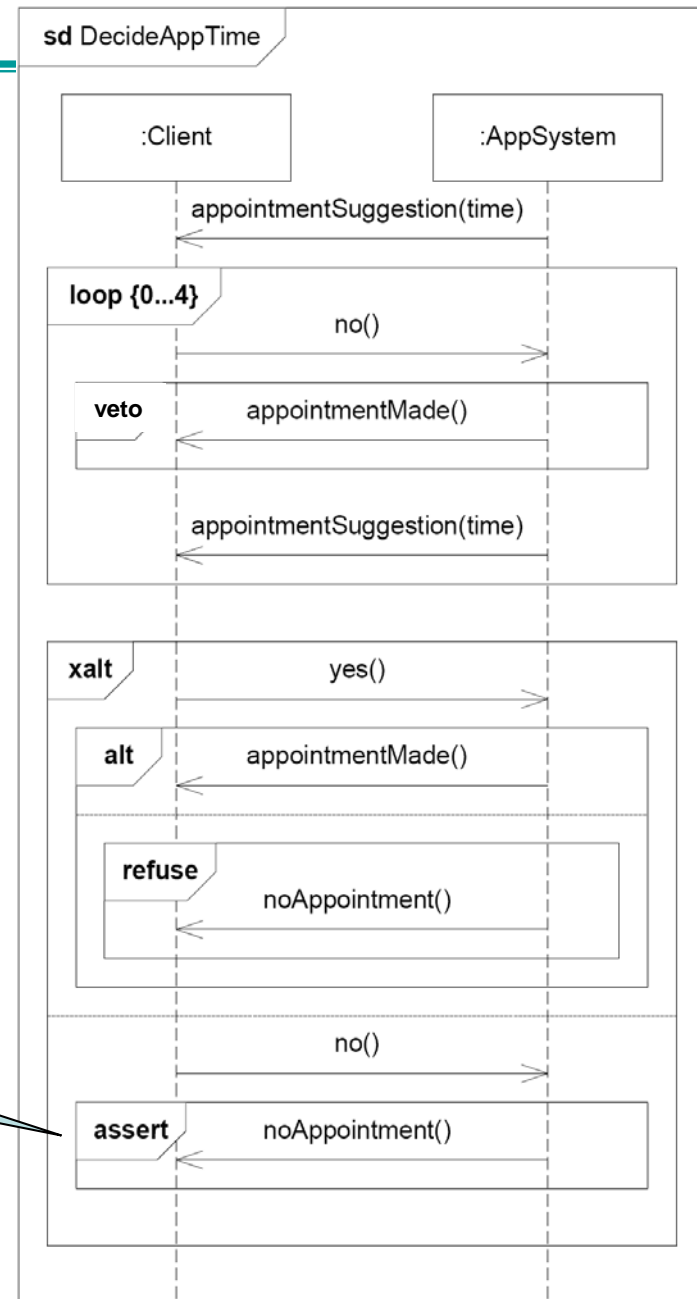
It is OK to do nothing between no() and appointmentSuggestion(time)

It is not OK to do nothing after yes()



when to use assert?

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
 - Use `assert` with caution!





The pragmatics of refining interactions



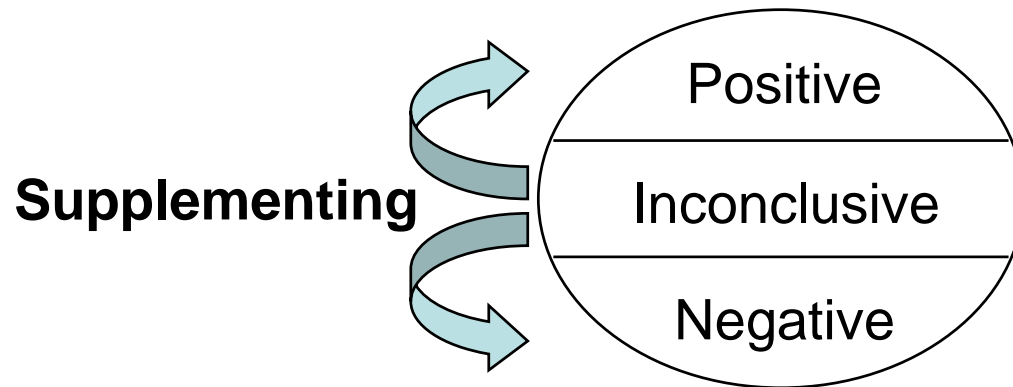
The use of supplementing

- 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



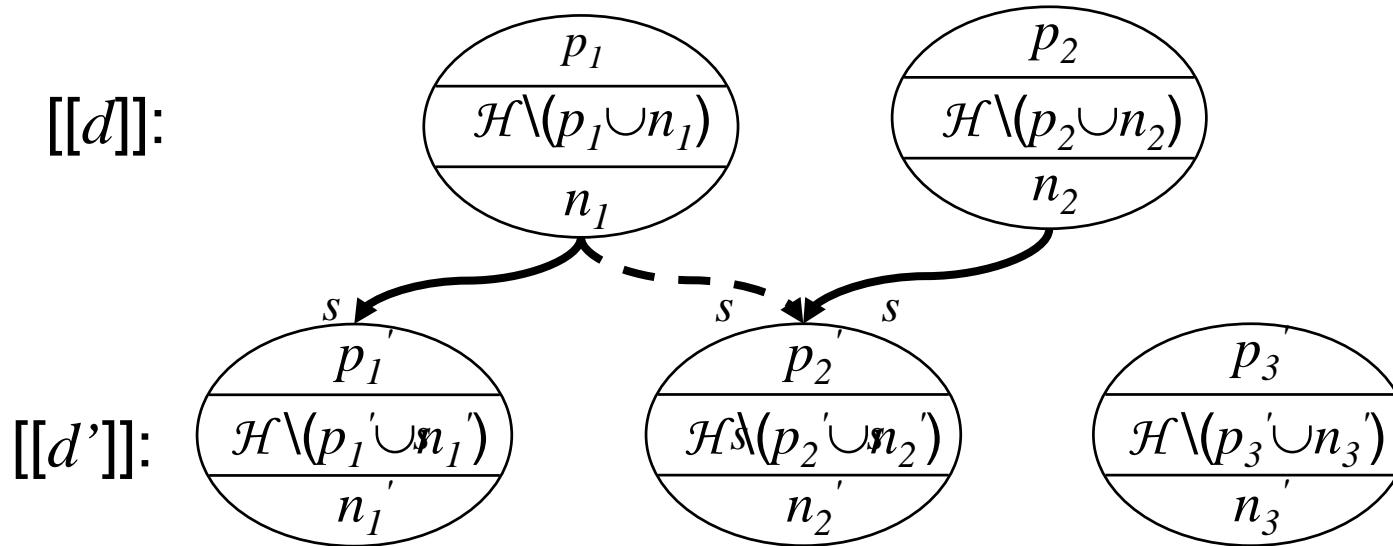
Supplementing of interaction obligations

- $(p, n) \rightsquigarrow_s (p', n') \stackrel{\text{def}}{=} p \subseteq p' \wedge n \subseteq n'$

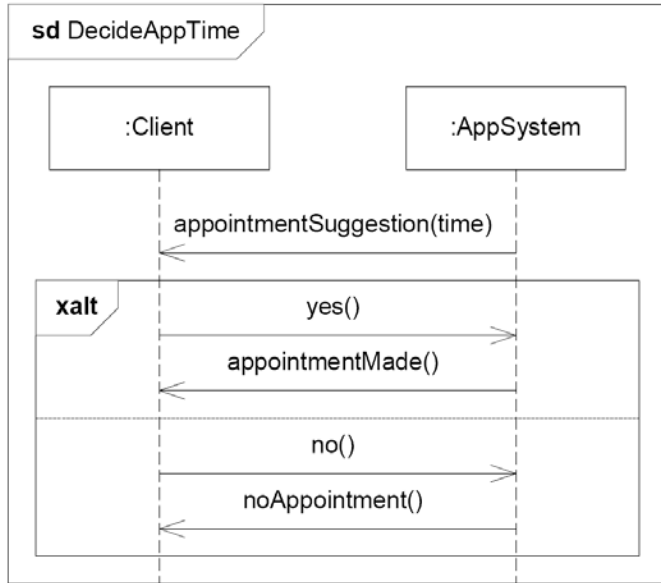


Supplementing of specifications

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

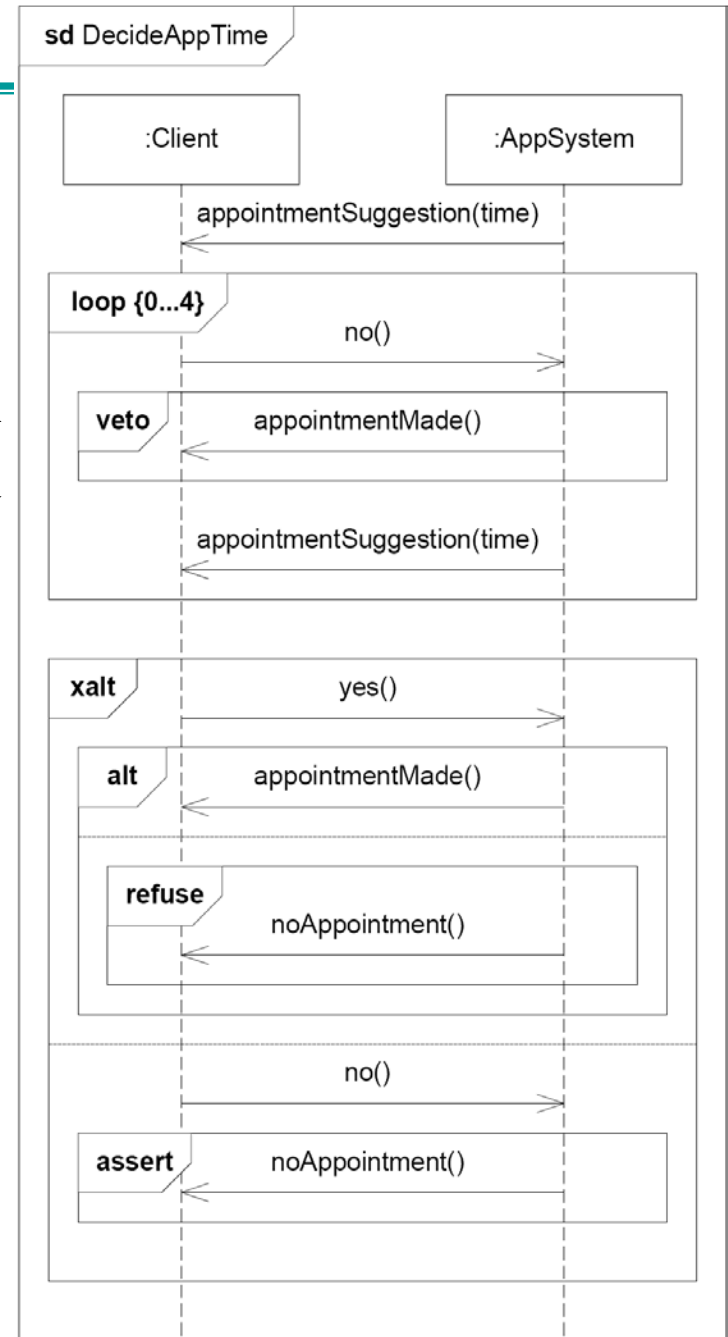


Example of supplementing



Positive

Negative



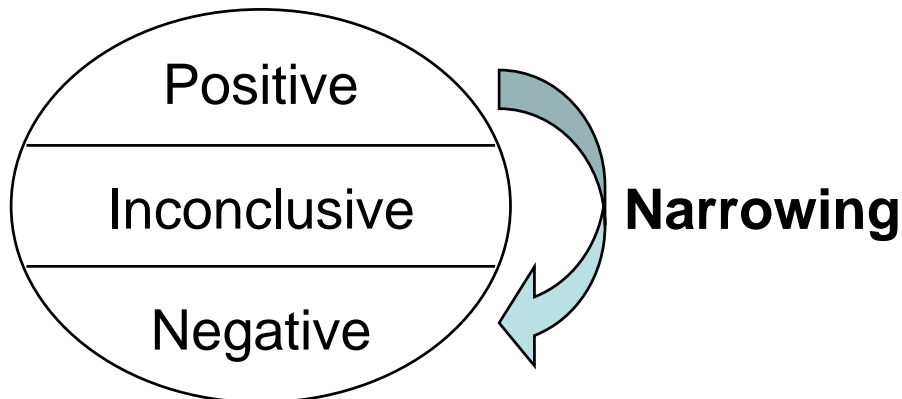
The pragmatics of supplementing

- 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

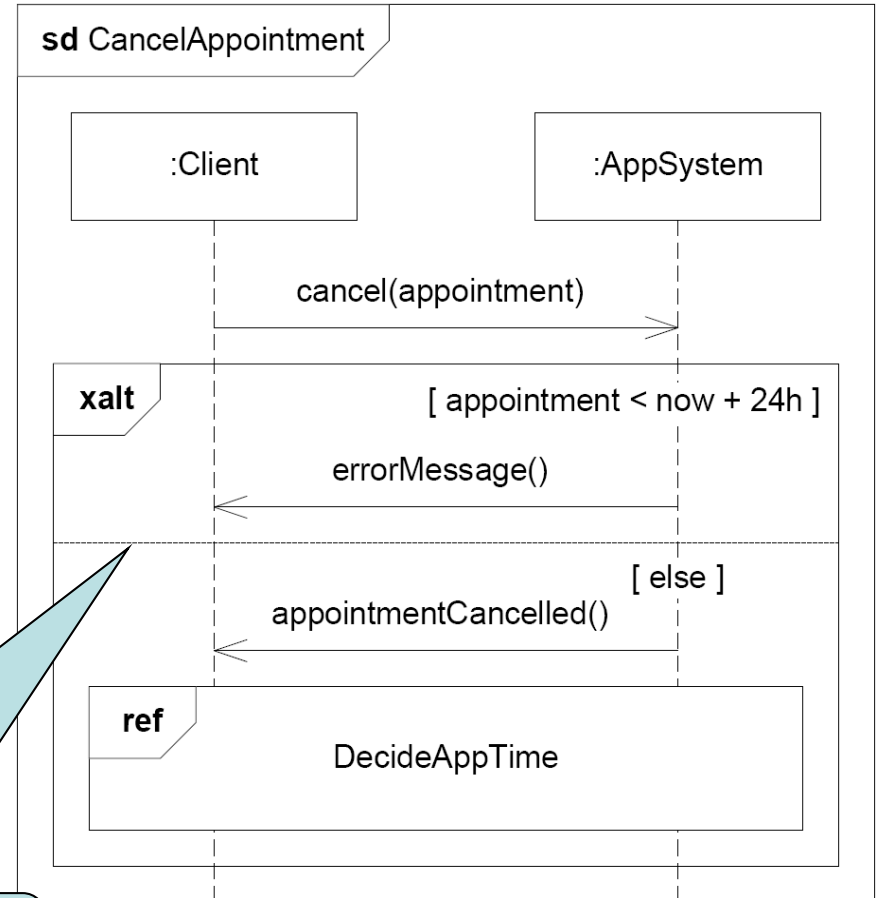
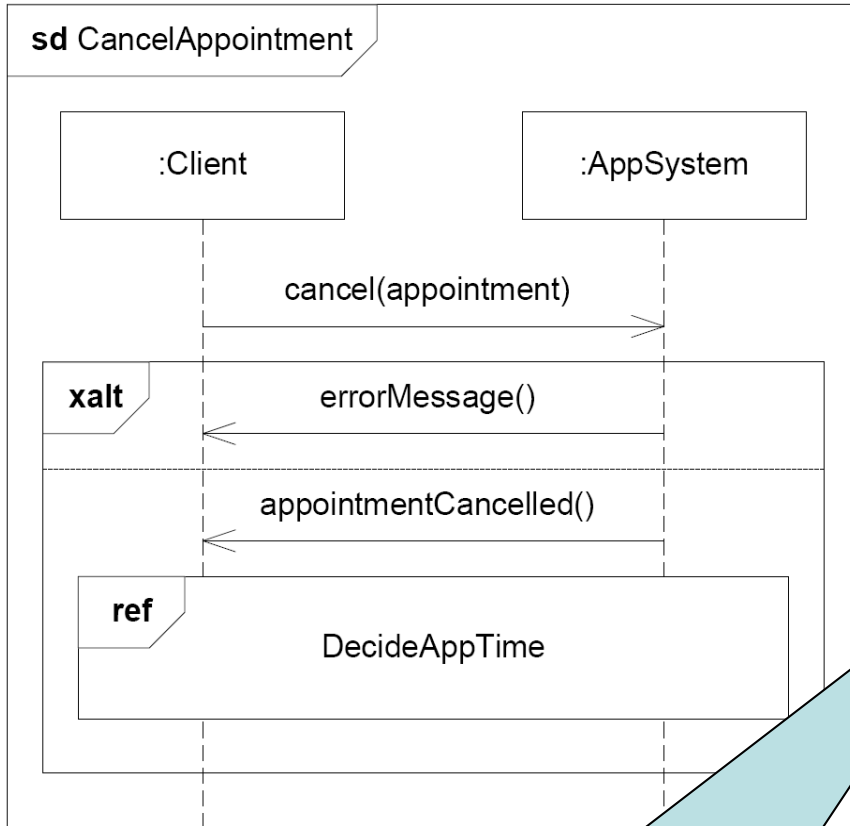


Narrowing

- 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
- $(p, n) \rightsquigarrow_n (p', n') \stackrel{\text{def}}{=} p' \subseteq p \wedge n' = n \cup (p \setminus p')$
- $d \rightsquigarrow_n d' \stackrel{\text{def}}{=} \forall o \in [[d]]: \exists o' \in [[d']]: o \rightsquigarrow_n o'$



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 positive 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

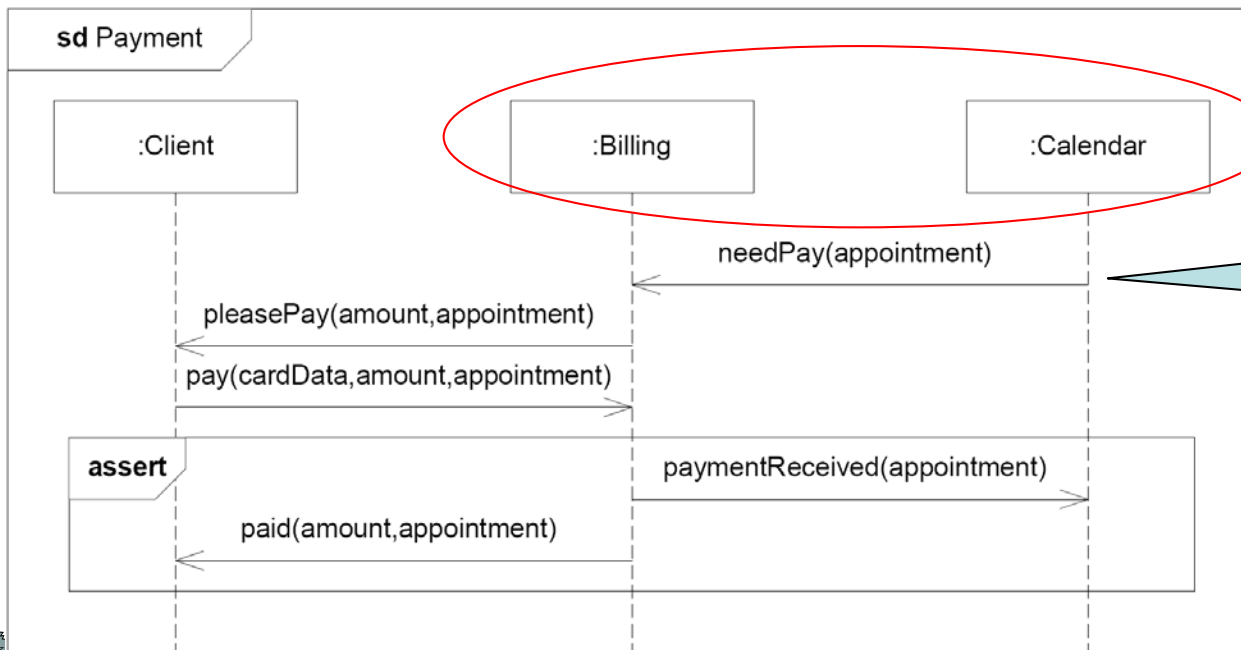
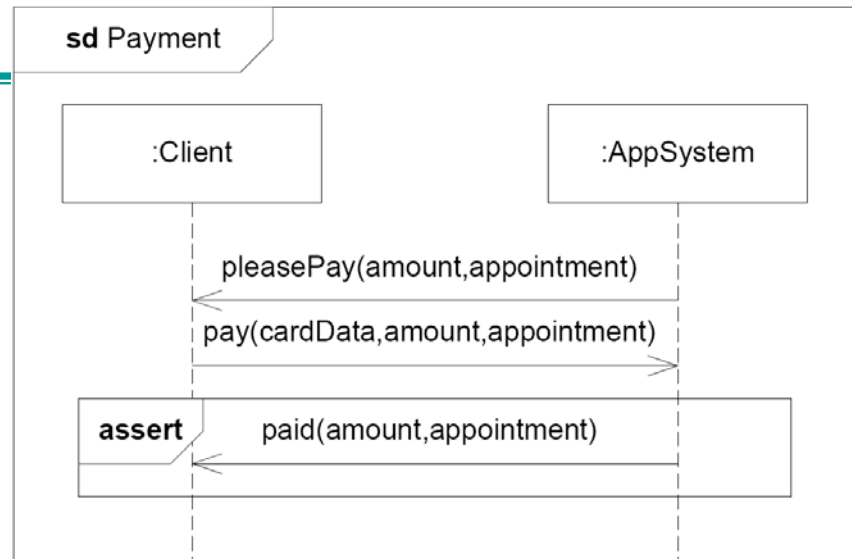


The use of detailing

- 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 occurring on a composed lifeline at the abstract level occur instead on one of the sub-component lifelines



Example of detailing



Components of
AppSystem

Internal
communication





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





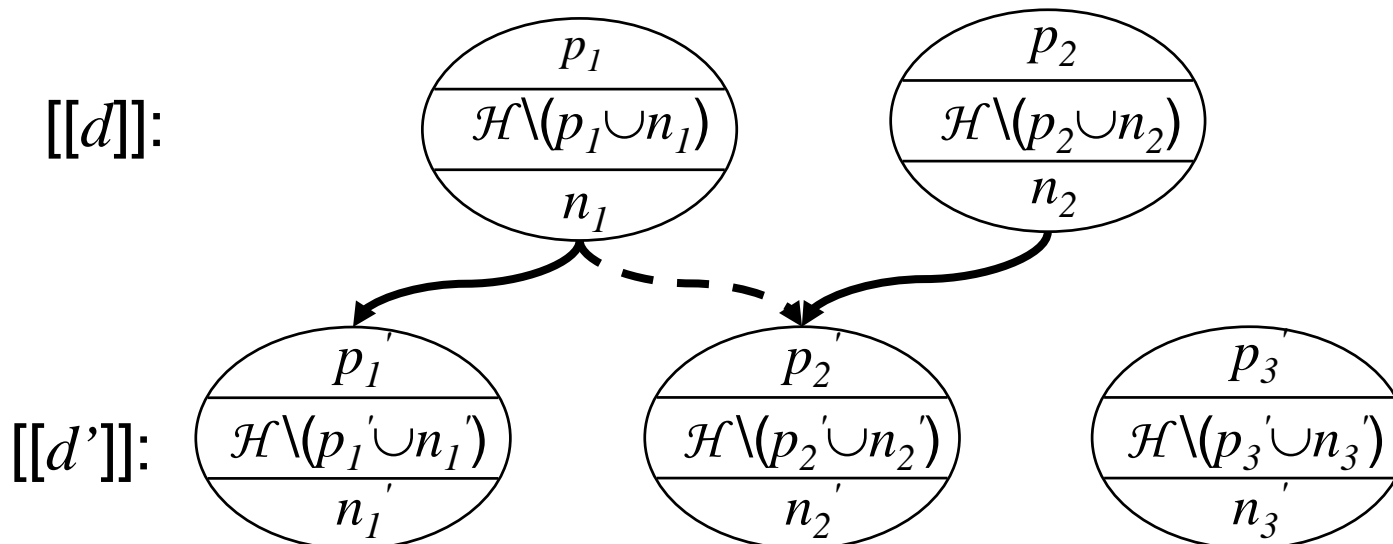
The use of general refinement

- A combination of supplementing, narrowing and detailing
 - (not necessarily all three)
- Allows all positive traces to become negative, while previously 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



General refinement (of sets of interaction obligations)

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



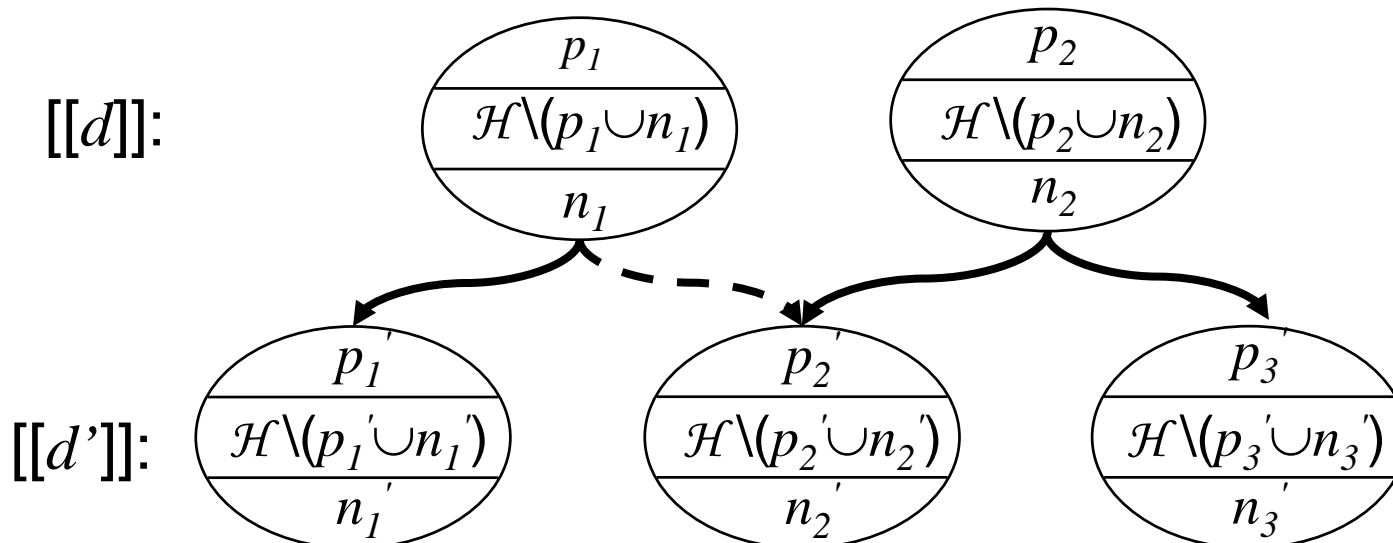
The pragmatics of general refinement

- 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 $[[d]]$





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 \rightsquigarrow is compositional if it is
 - reflexive: $d \rightsquigarrow d$
 - transitive: $d \rightsquigarrow d' \wedge d' \rightsquigarrow d'' \Rightarrow d \rightsquigarrow d''$
 - the operators `refuse`, `veto`, `alt`, `xalt` and `seq` are monotonic w.r.t. \rightsquigarrow :
 - $d \rightsquigarrow d' \Rightarrow \text{refuse } d \rightsquigarrow \text{refuse } d'$
 - $d \rightsquigarrow d' \Rightarrow \text{veto } d \rightsquigarrow \text{veto } d'$
 - $d_1 \rightsquigarrow d_1' \wedge d_2 \rightsquigarrow d_2' \Rightarrow d_1 \text{ alt } d_2 \rightsquigarrow d_1' \text{ alt } d_2'$
 - $d_1 \rightsquigarrow d_1' \wedge d_2 \rightsquigarrow d_2' \Rightarrow d_1 \text{ xalt } d_2 \rightsquigarrow d_1' \text{ xalt } d_2'$
 - $d_1 \rightsquigarrow d_1' \wedge d_2 \rightsquigarrow d_2' \Rightarrow d_1 \text{ seq } d_2 \rightsquigarrow d_1' \text{ seq } d_2'$
- 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 😊

