STAIRS from initial UML interactions to implementations

Ragnhild Kobro Runde
University of Oslo, Norway

Joint work with Øystein Haugen, Atle Refsdal and Ketil Stølen
Requirements to STAIRS

- Should allow specification of potential behaviour
  - Support for under-specification
- Should allow specification of mandatory behaviour
  - Support for information hiding (inherent non-determinism, unpredictability)
- Should allow specification of negative behaviour in addition to positive behaviour
  - Support for threat modeling
- Should capture the notion of refinement
- Should formalize incremental development
- Should support compositional analysis, verification and testing
Interactions and trace semantics

- Partial ordering of events:
  - The send event is ordered before the corresponding receive event.
  - Events on the same lifeline are ordered from the top and downwards.
- S specifies seven traces:
  \[
  \begin{align*}
  &< !x, ?x, !y, ?y, X_A, X_B > &< !x, !y, ?x, ?y, X_A, X_B > \\
  &< !x, ?x, !y, ?y, X_B, X_A > &< !x, !y, ?x, ?y, X_B, X_A > \\
  &< !x, ?x, !y, X_A, ?y, X_B > &< !x, !y, ?x, X_A, ?y, X_B > \\
  &< !x, !y, X_A, ?x, ?y, X_B > &< !x, !y, X_A, ?x, ?y, X_B >
  \end{align*}
  \]
Well-formedness

M, L  The set of all messages and lifelines

#h  The length of the trace h

h↓s  The trace h restricted to events in the set s

h↓l  The trace h restricted to events on the lifeline l

H denotes the set of all well-formed traces:

1. For all messages, the send event is ordered before the corresponding receive event:

\[ \forall h \in H: \forall m \in M: \forall i \in [1...#h]: h[i] = !m \Rightarrow
#(h[1...i]↓{!m}) > #(h[1...i]↓{?m}) \]

2. No events may take place on a lifeline that has terminated:

\[ \forall h \in H: \forall l \in L: \forall i \in [1...#(h↓l)-1]: (h↓l)[i] \neq X_l \]
STAIRS and refinement (simple case)

• An interaction specifies a set of positive and a set of negative traces:

- Positive
- Inconclusive
- Negative

Supplementing
Narrowing

• Assume \([ [ \text{d} ] ] = (p, n)\) and \([ [ \text{d'} ] ] = (p', n')\).
  \text{d'}\ is a refinement of \text{d}, \text{d} \leadsto \text{d'},\ iff
  \[ n \subseteq n' \quad \text{and} \quad p \subseteq p' \cup n' \]
Weak sequencing

- Assume $[[d_1]] = (p_1, n_1)$ and $[[d_2]] = (p_2, n_2)$
- $[[d_1 \text{ seq } d_2]] = (p_1, n_1) \succeq (p_2, n_2)
  = (p_1 \succeq p_2, (p_1 \succeq n_2) \cup (n_1 \succeq p_2) \cup (n_1 \succeq n_2))$
- $s_1 \succeq s_2 = \{ h \in H \mid \exists h_1 \in s_1, h_2 \in s_2 : \forall l \in L : h \downarrow l = h_1 \downarrow l + h_2 \downarrow l \}$

Positive:
$\langle \text{suggestion(time), yes(), appMade()} \rangle$

Negative:
$\langle \text{suggestion(time), yes(), noApp()} \rangle$

+ termination events
Alternative behaviour: alt

- Alternatives specified using alt represent underspecification.
- Assume
  \[
  [[d_1]] = (p_1, n_1) \\
  [[d_2]] = (p_2, n_2)
  \]
- alt specifies potential behaviour:
  \[
  [[d_1 \text{ alt } d_2]] = (p_1 \cup p_2, n_1 \cup n_2)
  \]
Example using alt: MakeAppointment

- Some examples of refinement:
  - Redefine needApp(hour) as negative (narrowing).
  - Define a new alternative needApp() as positive (supplementing).
Underspecification and non-determinism

• Underspecification: Several alternative behaviours are considered equivalent (serve the same purpose).

• Inherent non-determinism: Alternative behaviours that must all be possible for the implementation.

• These two should be described differently!
Alternative behaviour: xalt

- Assume
  
  \[
  [[ d_1 ]] = (p_1, n_1) \\
  [[ d_2 ]] = (p_2, n_2)
  \]

- xalt specifies mandatory behaviour:
  
  \[
  [[ d_1 \text{xalt} d_2 ]] \\
  = [[ d_1 ]] \cup [[ d_2 ]] \\
  = (p_1, n_1) \cup (p_2, n_2)
  \]
Using xalt

- xalt specifies mandatory alternatives, i.e. alternatives that must all be present in an implementation
- xalt models
  - inherent nondeterminism
  - different input-scenarios
  - alternative behaviour where the conditions for each one are abstracted away
xalt: different input-scenarios

sd DecideAppTime

:Client

suggestion(time)

xalt

yes()

appointmentMade()

no()

noAppointment()

:AppSystem

DecideAppTime: Client

suggestion(time): AppSystem

yes(): xalt

appointmentMade(): xalt

no(): xalt

noAppointment(): xalt
xalt: abstraction from conditions

sd CancelAppointment

:Client

:AppSystem

cancel(appointment)

xalt

errorMessage

appointmentCancelled

ref

DecideAppTime
Refinement (general case)

• An interaction obligation \( o' = (p', n') \) is a refinement of an interaction obligation \( o = (p, n) \) iff
  - \( n \subseteq n' \)
  - \( p \subseteq p' \cup n' \)

Positive

Inconclusive

Negative

Supplementing

Narrowing
Refinement contd.

• An interaction $d'$ is a refinement of an interaction $d$ iff
  \[ \forall o \in [[d]] : \exists o' \in [[d']] : o \sim o' \]

---

**Refinement contd.**

\[ d : \]

\[
\begin{array}{c}
  p_1 \\
  \mathcal{H}(p_1 \cup n_1) \\
  n_1 \\
  \end{array}
\]

\[
\begin{array}{c}
  p_2 \\
  \mathcal{H}(p_2 \cup n_2) \\
  n_2 \\
  \end{array}
\]

\[ d' : \]

\[
\begin{array}{c}
  p_1' \\
  \mathcal{H}(p_1' \cup n_1') \\
  n_1' \\
  \end{array}
\]

\[
\begin{array}{c}
  p_2' \\
  \mathcal{H}(p_2' \cup n_2') \\
  n_2' \\
  \end{array}
\]

\[
\begin{array}{c}
  p_3' \\
  \mathcal{H}(p_3' \cup n_3') \\
  n_3' \\
  \end{array}
\]

---
Refinement contd.

- An interaction $d'$ is a refinement of an interaction $d$ iff 
  \[ \forall o \in [[d]] : \exists o' \in [[d']] : o \sim o' \]

\[ \begin{align*}
  d: & \\
  p_1 & \quad H \backslash (p_1 \cup n_1) \\
  & \quad n_1 \\
  p_2 & \quad H \backslash (p_2 \cup n_2) \\
  & \quad n_2 \\

d': & \\
  p_1' & \quad H \backslash (p_1' \cup n_1') \\
  & \quad n_1' \\
  p_2' & \quad H \backslash (p_2' \cup n_2') \\
  & \quad n_2' \\
  p_3' & \quad H \backslash (p_3' \cup n_3') \\
  & \quad n_3'
\end{align*} \]

NOT VALID!
Refinement: example

sd DecideAppTime

:Client

suggestion(time)

:xalt

yes()
appointmentMade()

no()
noAppointment()

:AppSystem

suggestion(time)

xalt

yes()

alt

appointmentMade()

refuse

noAppointment()

no()

alt

noAppointment()

refuse

appointmentMade()
Properties of refinement

- The refinement operator \( \sim \rightarrow \) is:
  - Reflexive: \( d \sim \rightarrow d \)
  - Transitive: \( d \sim \rightarrow d' \land d' \sim \rightarrow d'' \Rightarrow d \sim \rightarrow d'' \)
  - Monotonic with respect to the main composition operators of interactions:
    \[ d_1 \sim \rightarrow d_1' \land d_2 \sim \rightarrow d_2' \Rightarrow d_1 \text{ op } d_2 \sim \rightarrow d_1' \text{ op } d_2' \]
Assert

- Considers only traces with events on the given lifelines.
- Prefix-related traces are still inconclusive.
- $[[\text{assert } d]] = \{ (p,n \cup H^{ll(d)} \setminus \text{related}(p, ll(d)) \mid (p,n) \in [[d]] \}$

$\text{related}(p,L) = \{ h \in H^L \mid \exists t \in p: \forall l \in L: (h \downarrow l \leq t \downarrow l \lor t \downarrow l \leq h \downarrow l) \}$

$ll(d)$  The lifelines in the diagram $d$
$q \leq r$  The sequence $q$ is a prefix of the sequence $r$
Assert: example

sd DecideAppTime

:Client

:AppSystem

assert

suggestion(time)

xalt

yes()

appointmentMade()

no()

noAppointment()
Limited refinement

• Assert restricts the possibility of adding new positive behaviour within one interaction obligation.

• Limited refinement restricts the possibility of adding new interaction obligations.

• An interaction $d'$ is a limited refinement of an interaction $d$ if $d \rightsquigarrow d'$ and also

$\forall o' \in [[d']] : \exists o \in [[d]] : o \rightsquigarrow o'$
Implementation

• Let I be an implementation.
• Let Tr(I) be the traces of I.
• From Tr(I) we construct a set of interaction obligations:
  \[ \{ ( \{ t \}, H^l \setminus \{ t \} ) \mid t \in \text{Tr}(I) \} \]
• An implementation I is a refinement of an interaction d iff I is a (limited) refinement of d.
Claims

- STAIRS allows specification of potential behaviour (underspecification)
- STAIRS allows specification of inherent nondeterminism
- STAIRS allows specification of negative behaviour in addition to positive behaviour
  - handles inconclusive behaviour
- STAIRS captures the notion of refinement
- STAIRS formalizes incremental development
- STAIRS supports compositional analysis, verification and testing