Software Design, Modelling and Analysis in UML

Lecture 19: Live Sequence Charts III

2013-01-23

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

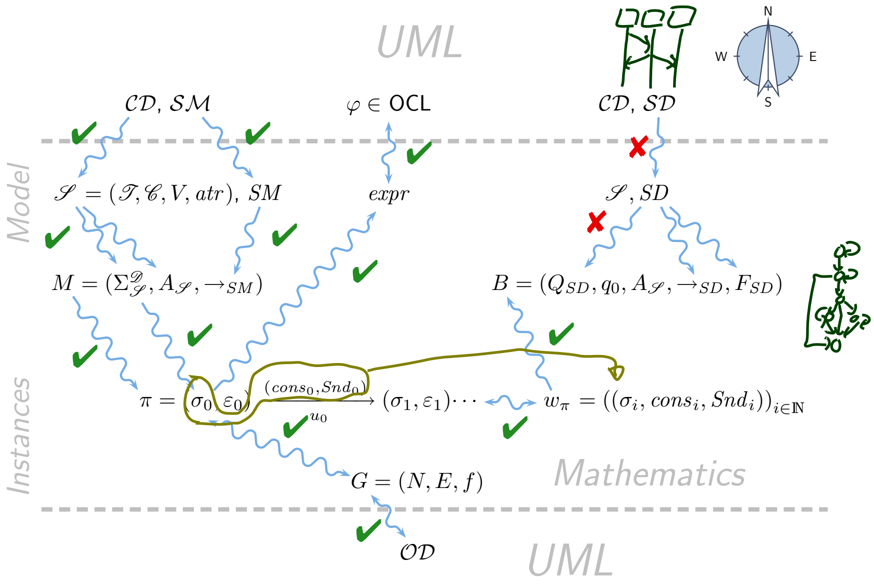
Last Lecture:

- Symbolic Büchi Automata (TBA) and its (accepted) language.
- Words of a model.

This Lecture:

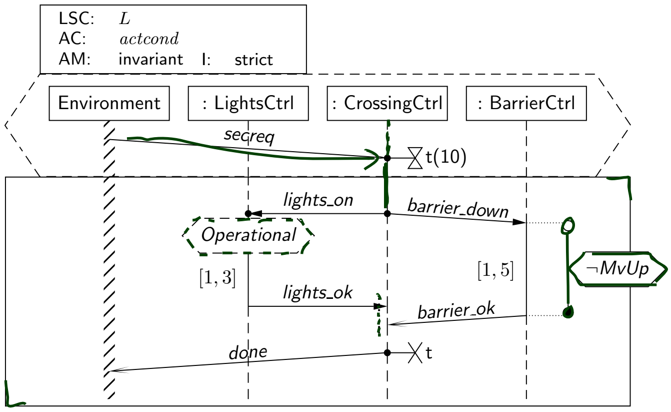
- **Educational Objectives:** Capabilities for following tasks/questions.
  - What does this LSC mean?
  - Are this UML model's state machines consistent with the interactions?
  - Please provide a UML model which is consistent with this LSC.
  - What is: activation, hot/cold condition, pre-chart, etc.?
- **Content:**
  - LSC abstract syntax.
  - LSC formal semantics.

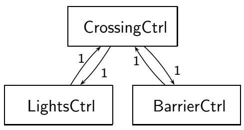
Course Map

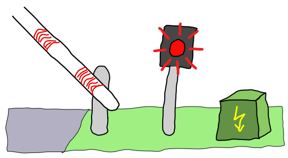


Live Sequence Charts Abstract Syntax

Example





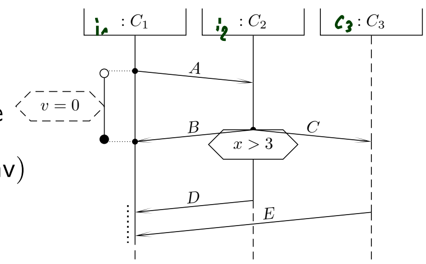


LSC Body: Abstract Syntax

Let $\Theta = \{\text{hot}, \text{cold}\}$. An **LSC body** is a tuple

$$(I, (\mathscr{L}, \preceq), \sim, \mathscr{S}, \text{Msg}, \text{Cond}, \text{LocInv})$$

- I is a finite set of **instance lines**,



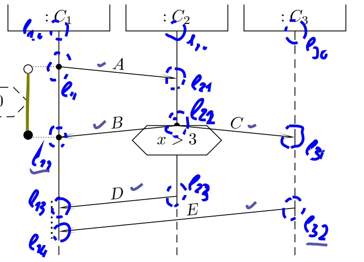
$I = \{i_1, i_2, c_3\}$

LSC Body: Abstract Syntax

Let $\Theta = \{\text{hot}, \text{cold}\}$. An **LSC body** is a tuple

$$(I, (\mathscr{L}, \preceq), \sim, \mathscr{S}, \mathsf{Msg}, \mathsf{Cond}, \mathsf{LocInv})$$

- I is a finite set of **instance lines**,
- $(\mathscr{L}, \preceq)$ is a finite, non-empty, **partially ordered** set of **locations**; each $l \in \mathscr{L}$ is associated with a temperature $\theta(l) \in \Theta$ and an instance line $i_l \in I$,



$\mathscr{L} = \{l_{10}, .., l_{14}, l_{20}, .., l_{23}, l_{30}, .., l_{32}\}$

$\preceq$: $l_{10} \prec l_{11}$, $l_{11} \prec l_{12}$, $l_{12} \prec l_{13}$, $l_{13} \prec l_{14}$

$l_{22} \prec l_{12}$, $l_{11} \prec l_{21}$, $l_{22} \prec l_{31}$,

$l_{20} \prec l_{21} \prec l_{22} \prec l_{23}$ $\quad$ $l_{23} \prec l_{31}$,

$l_{30} \prec l_{31} \prec l_{32}$ $\quad$ $l_{32} \prec l_{14}$

$\Theta(l_{10}) = \text{hot}$

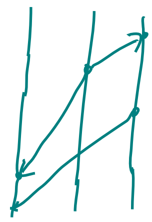
$\Theta(l_{31}) : \text{cold}$

$\Theta(l_{22}) = \text{hot}$

$\Theta(l_{23}) = \text{cold}$

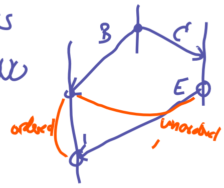
$\Theta(l_{13}) = \text{cold}$ } one temp.

$\Theta(l_{14}) = \text{cold}$ } for coregion



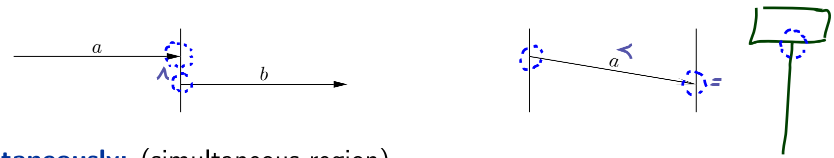
$l_{32} \succeq l_{11}$? YES

$l_{12} \not\preceq l_{32}$? NO

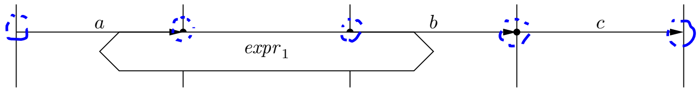


Recall: Intuitive Semantics

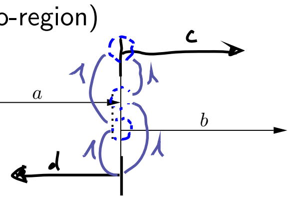
(i) **Strictly After:**



(ii) **Simultaneously:** (simultaneous region)

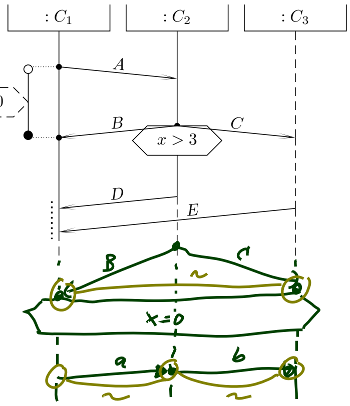


(iii) **Explicitly Unordered:** (co-region)



Intuition: A computation path **violates** an LSC if the occurrence of some events doesn't adhere to the partial order obtained as the **transitive closure** of (i) to (iii).

LSC Body: Abstract Syntax

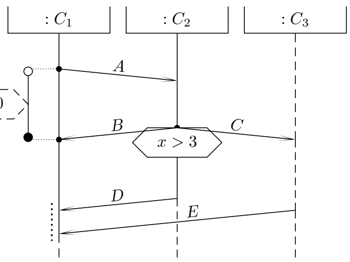


Let $\Theta = \{\text{hot}, \text{cold}\}$. An **LSC body** is a tuple

$$(I, (\mathscr{L}, \preceq), \sim, \mathscr{S}, \text{Msg}, \text{Cond}, \text{LocInv})$$

- I is a finite set of **instance lines**,
- $(\mathscr{L}, \preceq)$ is a finite, non-empty, **partially ordered** set of **locations**; each $l \in \mathscr{L}$ is associated with a temperature $\theta(l) \in \Theta$ and an instance line $i_l \in I$,
- $\sim \subseteq \mathscr{L} \times \mathscr{L}$ is an **equivalence relation** on locations, the **simultaneity** relation,

LSC Body: Abstract Syntax



Let $\Theta = \{\text{hot}, \text{cold}\}$. An **LSC body** is a tuple

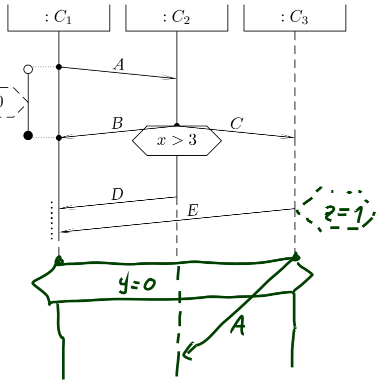
$$(I, (\mathscr{L}, \preceq), \sim, \mathscr{S}, \text{Msg}, \text{Cond}, \text{LocInv})$$

- I is a finite set of **instance lines**,
- $(\mathscr{L}, \preceq)$ is a finite, non-empty, **partially ordered** set of **locations**; each $l \in \mathscr{L}$ is associated with a temperature $\theta(l) \in \Theta$ and an instance line $i_l \in I$,
- $\sim \subseteq \mathscr{L} \times \mathscr{L}$ is an **equivalence relation** on locations, the **simultaneity** relation,

- $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$ is a signature,
- $\text{Msg} \subseteq \mathscr{L} \times \mathscr{E} \times \mathscr{L}$ is a set of **asynchronous messages** with $(l, b, l') \in \text{Msg}$ only if $l \preceq l'$,
  Not: **instantaneous messages** —
  could be linked to method/operation calls.

$\text{Msg} = \{ (l_{11}, A, l_{12}),$
$(l_{13}, B, l_{02}),$
$(l_{13}, C, l_{21}),$
$(l_{23}, D, l_{03}),$
$(l_{12}, E, l_{04}) \}$

LSC Body: Abstract Syntax



Let $\Theta = \{\text{hot}, \text{cold}\}$. An **LSC body** is a tuple

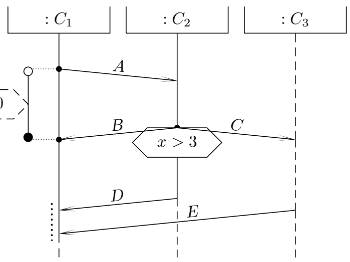
$$(I, (\mathscr{L}, \preceq), \sim, \mathscr{S}, \text{Msg}, \text{Cond}, \text{LocInv})$$

- I is a finite set of **instance lines**,
- $(\mathscr{L}, \preceq)$ is a finite, non-empty, **partially ordered** set of **locations**; each $l \in \mathscr{L}$ is associated with a temperature $\theta(l) \in \Theta$ and an instance line $i_l \in I$,
- $\sim \subseteq \mathscr{L} \times \mathscr{L}$ is an **equivalence relation** on locations, the **simultaneity** relation,
- $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$ is a signature,
- $\text{Msg} \subseteq \mathscr{L} \times \mathscr{E} \times \mathscr{L}$ is a set of **asynchronous messages** with $(l, b, l') \in \text{Msg}$ only if $l \preceq l'$,
Not: **instantaneous messages** —
could be linked to method/operation calls.
- $\text{Cond} \subseteq (2^{\mathscr{L}} \setminus \emptyset) \times Expr_{\mathscr{S}} \times \Theta$ is a set of **conditions** where $Expr_{\mathscr{S}}$ are OCL expressions over $W = I \cup \{self\}$ with $(L, expr, \theta) \in \text{Cond}$ only if $l \sim l'$ for all $l, l' \in L$,

$\text{Cond} = \{ (\{l_{12}\}, x>3, hot), (\{l_{22}\}, z=1, cold), (\{l_{05}, l_{23}\}, y=0, hot) \}$

– 19 – 2013-01-23 – Slscasyn –

LSC Body: Abstract Syntax



Let $\Theta = \{\text{hot}, \text{cold}\}$. An **LSC body** is a tuple

$$(I, (\mathscr{L}, \preceq), \sim, \mathscr{S}, \text{Msg}, \text{Cond}, \text{LocInv})$$

- I is a finite set of **instance lines**,
- $(\mathscr{L}, \preceq)$ is a finite, non-empty, **partially ordered** set of **locations**; each $l \in \mathscr{L}$ is associated with a temperature $\theta(l) \in \Theta$ and an instance line $i_l \in I$,
- $\sim \subseteq \mathscr{L} \times \mathscr{L}$ is an **equivalence relation** on locations, the **simultaneity** relation,
- $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$ is a signature,
- $\text{Msg} \subseteq \mathscr{L} \times \mathscr{E} \times \mathscr{L}$ is a set of **asynchronous messages** with $(l, b, l') \in \text{Msg}$ only if $l \preceq l'$,
Not: **instantaneous messages** —
could be linked to method/operation calls.
- $\text{Cond} \subseteq (2^{\mathscr{L}} \setminus \emptyset) \times Expr_{\mathscr{S}} \times \Theta$ is a set of **conditions** where $Expr_{\mathscr{S}}$ are OCL expressions over $W = I \cup \{self\}$ with $(L, expr, \theta) \in \text{Cond}$ only if $l \sim l'$ for all $l, l' \in L$,
- $\text{LocInv} \subseteq \mathscr{L} \times \{\circ, \bullet\} \times Expr_{\mathscr{S}} \times \Theta \times \mathscr{L} \times \{\circ, \bullet\}$ is a set of **local invariants**,

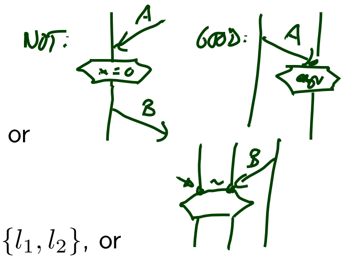
$\text{LocInv} = \{ (l_{01}, \circ, v=0, cold, l_{02}, \bullet) \}$

– 19 – 2013-01-23 – Slscasyn –

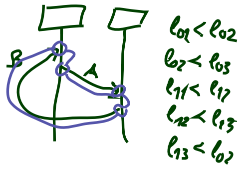
Well-Formedness

Bondedness/no floating conditions: (could be relaxed a little if we wanted to)

- For each location $l \in \mathscr{L}$, **if** l is the location of
  - a **condition**, i.e.
    $$\exists (L, expr, \theta) \in \mathsf{Cond} : l \in L, \text{ or}$$
  - a **local invariant**, i.e.
    $$\exists (l_1, i_1, expr, \theta, l_2, i_2) \in \mathsf{LocInv} : l \in \{l_1, l_2\}, \text{ or}$$

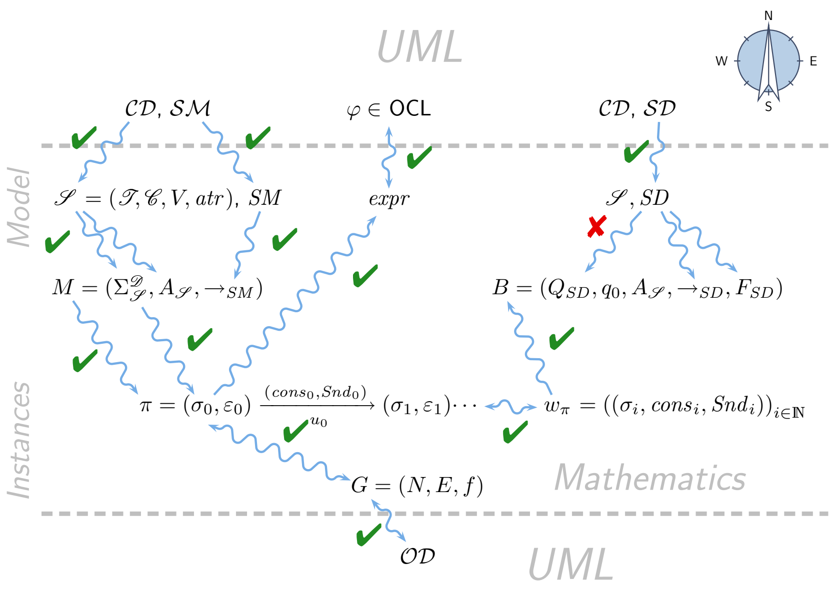


  then there is a location l' **equivalent** to l, i.e. $l \sim l'$, which is the location of
  - an **instance head**, i.e. l' is minimal wrt. $\preceq$, or
  - a **message**, i.e.
    $$\exists (l_1, b, l_2) \in \mathsf{Msg} : l \in \{l_1, l_2\}.$$



Note: if messages in a chart are **cyclic**, then there doesn't exist a partial order (so such charts **don't even have** an abstract syntax).

Course Map



Live Sequence Charts Semantics

TBA-based Semantics of LSCs

Plan:

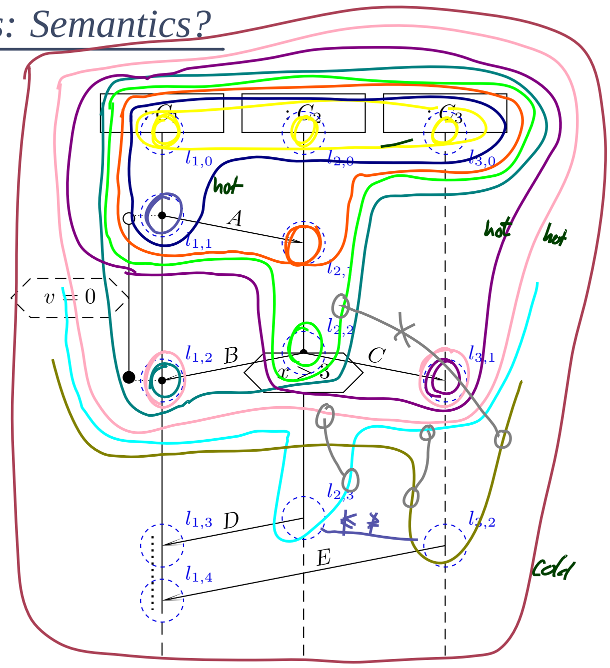
- Given an LSC L with body

$$(I, (\mathscr{L}, \preceq), \sim, \mathscr{S}, \mathsf{Msg}, \mathsf{Cond}, \mathsf{LocInv}),$$

- construct a TBA $\mathcal{B}_L$, and
- define $\mathcal{L}(L)$ **in terms of** $\mathcal{L}(\mathcal{B}_L)$,
  in particular taking activation condition and activation mode into account.

- Then $\mathcal{M} \models L$ (universal) if and only if $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(L)$.

Examples: Semantics?



Formal LSC Semantics: It's in the Cuts!

Definition.
Let $(I, (\mathscr{L}, \preceq), \sim, \mathscr{S}, \mathsf{Msg}, \mathsf{Cond}, \mathsf{LocInv})$ be an LSC body.
A non-empty set $\emptyset \neq C \subseteq \mathscr{L}$ is called a **cut** of the LSC body iff

- it is **downward closed**, i.e.

$$\forall\, l, l' : l' \in C \wedge l \preceq l' \implies l \in C,$$

- it is **closed** under **simultaneity**, i.e.

$$\forall\, l, l' : l' \in C \wedge l \sim l' \implies l \in C, \text{ and}$$

- it comprises at least **one location per instance line**, i.e.

$$\forall\, i \in I\ \exists\, l \in C : i_l = i.$$

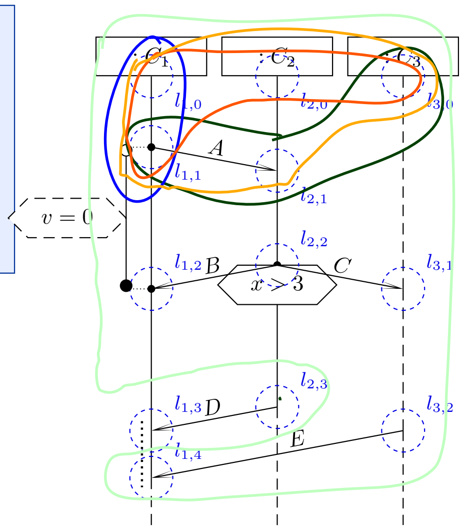
A cut C is called **hot**, denoted by $\theta(C) = \mathsf{hot}$, if and only if at least one of its maximal elements is hot, i.e. if

$$\exists\, l \in C : \theta(l) = \mathsf{hot} \wedge \nexists\, l' \in C : l \prec l'$$

Otherwise, C is called **cold**, denoted by $\theta(C) = \mathsf{cold}$.

Examples: Cut or Not Cut? Hot/Cold?

(i) **non-empty** set $\emptyset \neq C \subseteq \mathscr{L}$,
(ii) **downward closed**, i.e.
$\forall\, l, l' : l' \in C \wedge l \preceq l' \implies l \in C$
(iii) **closed** under **simultaneity**, i.e.
$\forall\, l, l' : l' \in C \wedge l \sim l' \implies l \in C$
(iv) at least **one location per instance line**, i.e.
$\forall\, i \in I\; \exists\, l \in C : i_l = i$,



- $C_0 = \emptyset$ NO
- $C_1 = \{l_{1,0}, l_{2,0}, l_{3,0}\}$ ✓
- $C_2 = \{l_{1,1}, l_{2,1}, l_{3,0}\}$ NO
- $C_3 = \{l_{1,0}, l_{1,1}\}$ NO
- $C_4 = \{l_{1,0}, l_{1,1}, l_{2,0}, l_{3,0}\}$ ✓
- $C_5 = \{l_{1,0}, l_{1,1}, l_{2,0}, l_{2,1}, l_{3,0}\}$ ✓
- $C_6 = \mathscr{L} \setminus \{l_{1,3}, l_{2,3}\}$ ✓
- $C_7 = \mathscr{L}$ ✓

A Successor Relation on Cuts

The partial order of $(\mathscr{L}, \preceq)$ and the simultaneity relation "$\sim$" induce a **direct successor relation** on cuts of $\mathscr{L}$ as follows:

Definition. Let $C, C' \subseteq \mathscr{L}$ bet cuts of an LSC body with locations $(\mathscr{L}, \preceq)$ and messages Msg.
C' is called **direct successor** of C **via fired-set** F, denoted by $\boxed{C \rightsquigarrow_F C'}$, if and only if

- $F \neq \emptyset$,
- $C' \setminus C = F$,
- for each message reception in F, the corresponding sending is already in C,
$$\forall\, (l, E, l') \in \mathsf{Msg} : l' \in F \implies l \in C, \text{ and}$$
- locations in F, that lie on the same instance line, are pairwise unordered, i.e.
$$\forall\, l, l' \in F : l \neq l' \wedge i_l = i_{l'} \implies l \not\preceq l' \wedge l' \not\preceq l$$

Properties of the Fired-set

$C \rightsquigarrow_F C'$ if and only if
- $F \neq \emptyset$,
- $C' \setminus C = F$,
- $\forall\,(l, E, l') \in \mathsf{Msg} : l' \in F \implies l \in C$, and
- $\forall\, l, l' \in F : l \neq l' \wedge i_l = i_{l'} \implies l \not\preceq l' \wedge l' \not\preceq l$

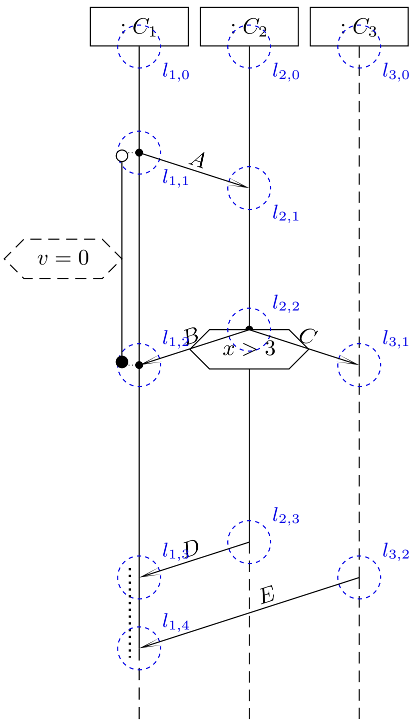
- **Note**: F is closed under simultaneity.
- **Note**: locations in F are direct $\preceq$-successors of locations in C, i.e.

$$\forall\, l' \in F\ \exists\, l \in C : l \prec l' \wedge \nexists\, l'' \in C : l' \prec l'' \prec l$$

Successor Cut Examples

(i) $F \neq \emptyset$, (ii) $C' \setminus C = F$,
(iii) $\forall\,(l, E, l') \in \mathsf{Msg} : l' \in F \implies l \in C$, and
(iv) $\forall\, l, l' \in F : l \neq l' \wedge i_l = i_{l'} \implies l \not\preceq l' \wedge l' \not\preceq l$

see Slide 12



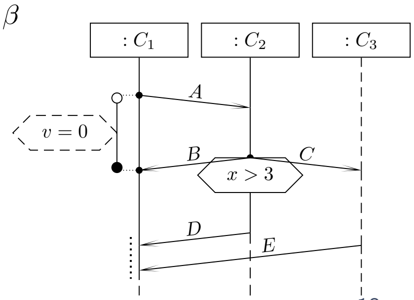
Idea: Accept Timed Words by Advancing the Cut

- Let $w = (\sigma_0, cons_0, Snd_0), (\sigma_1, cons_1, Snd_1), (\sigma_2, cons_2, Snd_2), \dots$ be a word of a UML model and β a valuation of $I \cup \{self\}$.
- **Intuitively** (and for now **disregarding** cold conditions),
  an LSC body $(I, (\mathscr{L}, \preceq), \sim, \mathscr{S}, \mathsf{Msg}, \mathsf{Cond}, \mathsf{LocInv})$
  is **supposed** to **accept** w if and only if there exists a sequence

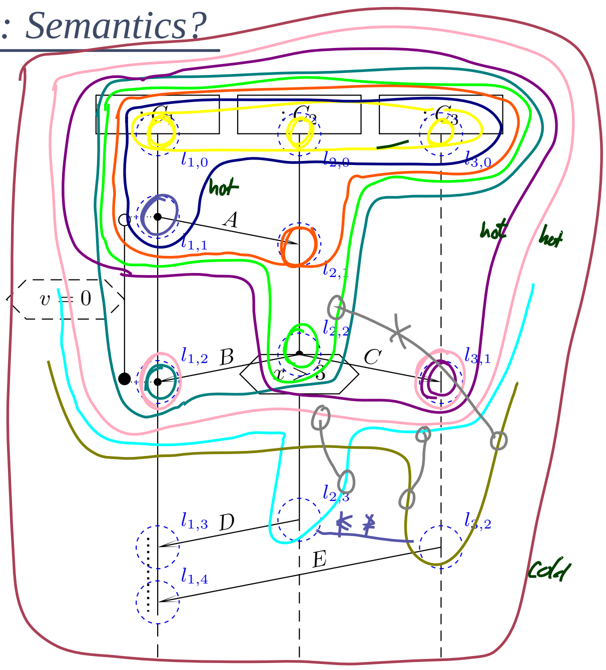
$$C_0 \rightsquigarrow_{F_1} C_1 \rightsquigarrow_{F_2} C_2 \cdots \rightsquigarrow_{F_n} C_n$$

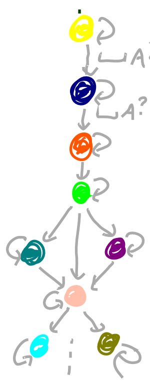
  and indices $0 = i_0 < i_1 < \cdots < i_n$ such that for all $0 \leq j < n$,

  - for all $i_j \leq k < i_{j+1}$, $(\sigma_k, cons_k, Snd_k)$, β satisfies the **hold condition** of C_j,
  - $(\sigma_{i_j}, cons_{i_j}, Snd_{i_j})$, β satisfies the **transition condition** of F_j,
  - C_n is cold,
  - for all $i_n < k$, $(\sigma_k, cons_{i_j}, Snd_{i_j})$, β satisfies the **hold condition** of C_n.



Examples: Semantics?





Language of LSC Body

The **language** of the body

$$(I, (\mathscr{L}, \preceq), \sim, \mathscr{S}, \mathsf{Msg}, \mathsf{Cond}, \mathsf{LocInv})$$

of LSC L is the language of the TBA

$$\mathcal{B}_L = (Expr_{\mathcal{B}}(X), X, Q, q_{ini}, \rightarrow, Q_F)$$

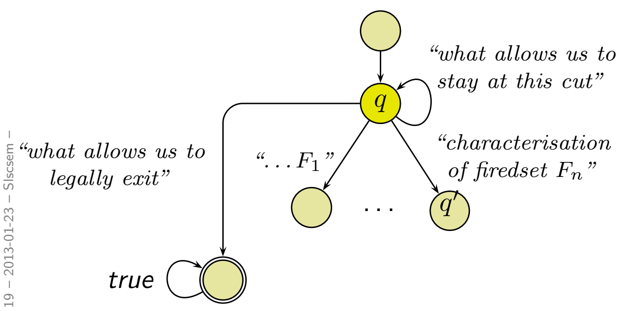
with

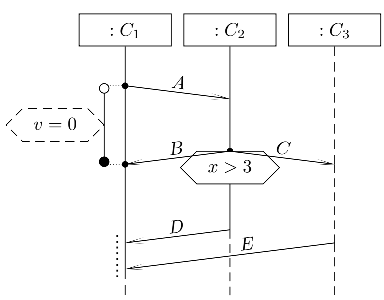
- $Expr_{\mathcal{B}}(X) = Expr_{\mathscr{S}}(\mathscr{S}, X)$
- Q is the set of cuts of $(\mathscr{L}, \preceq)$, q_{ini} is the **instance heads** cut,
- $F = \{C \in Q \mid \theta(C) = \mathsf{cold}\}$ is the set of cold cuts of $(\mathscr{L}, \preceq)$,
- $\rightarrow$ as defined in the following, consisting of
  - **loops** (q, ψ, q),
  - **progress transitions** (q, ψ, q') corresponding to $q \rightsquigarrow_F q'$, and
  - **legal exits** $(q, \psi, \mathscr{L})$.

Language of LSC Body: Intuition

$\mathcal{B}_L = (Expr_{\mathcal{B}}(X), X, Q, q_{ini}, \rightarrow, Q_F)$ with

- $Expr_{\mathcal{B}}(X) = Expr_{\mathscr{S}}(\mathscr{S}, X)$
- Q is the set of cuts of $(\mathscr{L}, \preceq)$, q_{ini} is the **instance heads** cut,
- $F = \{C \in Q \mid \theta(C) = \mathsf{cold}\}$ is the set of cold cuts,
- $\rightarrow$ consists of
  - **loops** (q, ψ, q),
  - **progress transitions** (q, ψ, q') corresponding to $q \rightsquigarrow_F q'$, and
  - **legal exits** $(q, \psi, \mathscr{L})$.





Step I: Only Messages

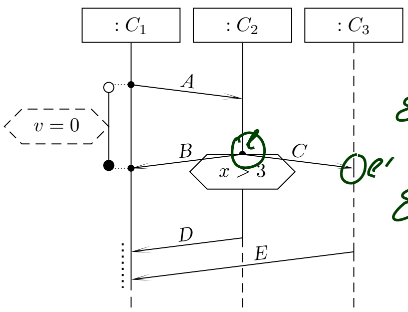
Some Helper Functions

- **Message-expressions of a location**:

$$\mathscr{E}(l) := \{E^!_{i_l, i_{l'}} \mid (l, E, l') \in \mathsf{Msg}\} \cup \{E^?_{i_{l'}, i_l} \mid (l', E, l) \in \mathsf{Msg}\},$$

$$\mathscr{E}(\{l_1, \dots, l_n\}) := \mathscr{E}(l_1) \cup \dots \cup \mathscr{E}(l_n).$$

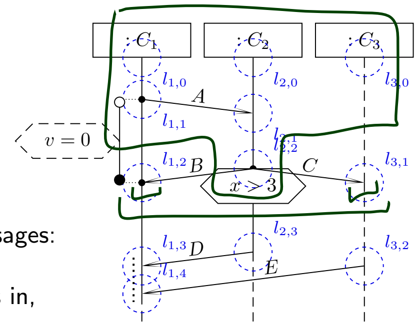
$$\bigvee \emptyset := \textit{true}; \bigvee \{E1^!_{i_{11}, i_{12}}, \dots Fk^?_{i_{k1}, i_{k2}}, \dots\} := \bigvee_{1 \le j < k} Ej^!_{i_{j1}, i_{j2}} \vee \bigvee_{k \le j} Fj^?_{i_{j1}, i_{j2}}$$



$\mathcal{E}(l) = \{B^!_{i_2, i_1}, C^!_{i_2, i_3}\}$

$\mathcal{E}(l') = \{C^?_{i_2, i_3}\}$

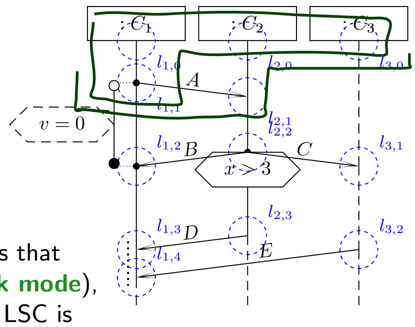
Loops



- How long may we **legally** stay at a cut q?
- **Intuition**: those $(\sigma_i, cons_i, Snd_i)$ are allowed to fire the self-loop (q, ψ, q) where
  - $cons_i \cup Snd_i$ comprises only irrelevant messages:
    - **weak mode**: no message from a direct successor cut is in,
    - **strict mode**: no message occurring in the LSC is in,
  - σ_i satisfies the local invariants active at q

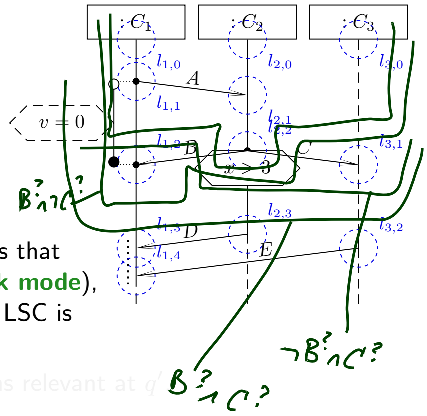
  And nothing else.
- **Formally**: Let $F := F_1 \cup \cdots \cup F_n$ be the union of the firedsets of q.
  - $\psi := \underbrace{\neg(\bigvee \mathscr{E}(F))}_{= \mathit{true} \text{ if } F = \emptyset} \wedge \bigwedge \psi(q).$

Progress



- When do we move from q to q'?
- **Intuition**: those $(\sigma_i, cons_i, Snd_i)$ fire the progress transition (q, ψ, q') for which there exists a firedset F such that $q \rightsquigarrow_F q'$ and
  - $cons_i \cup Snd_i$ comprises exactly the messages that distinguish F from other firedsets of q (**weak mode**), and in addition no message occurring in the LSC is in $cons_i \cup Snd_i$ (**strict mode**),
  - σ_i satisfies the local invariants and conditions relevant at q'.

Progress



- When do we move from q to q'?
- **Intuition**: those $(\sigma_i, cons_i, Snd_i)$ fire the progress transition (q, ψ, q') for which there exists a firedset F such that $q \rightsquigarrow_F q'$ and
  - $cons_i \cup Snd_i$ comprises exactly the messages that distinguish F from other firedsets of q (**weak mode**), and in addition no message occurring in the LSC is in $cons_i \cup Snd_i$ (**strict mode**),
  - σ_i satisfies the local invariants and conditions relevant at q'.
- **Formally**: Let $F, F_1, \dots, F_n$ be the firedsets of q and let $q \rightsquigarrow_F q'$ (unique).
  - $\psi := \underbrace{\bigwedge \mathscr{E}(F)} \wedge \underbrace{\neg(\bigvee(\mathscr{E}(F_1) \cup \dots \cup \mathscr{E}(F_n)) \setminus \mathscr{E}(F))} \wedge \bigwedge \psi(q, q')$.

Step II: Conditions and Local Invariants

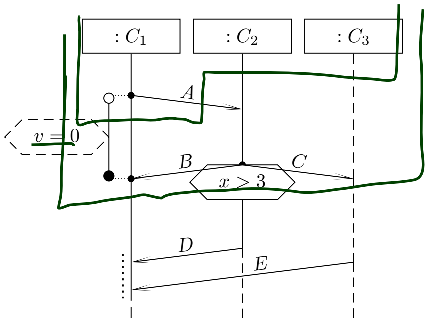
Some More Helper Functions

- **Constraints** relevant **at** cut q:

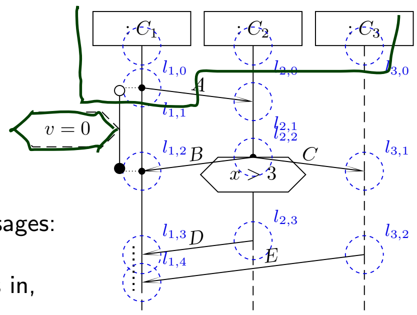
$$\psi_\theta(q) = \{\psi \mid \exists\, l \in q, l' \notin q \mid (l, \psi, \theta, l') \in \mathsf{LocInv} \vee (l', \psi, \theta, l) \in \mathsf{LocInv}\},$$

$$\psi(q) = \psi_{\mathsf{hot}}(q) \cup \psi_{\mathsf{cold}}(q)$$

$$\bigwedge \emptyset := \mathit{false}; \quad \bigwedge\{\psi_1, \dots, \psi_n\} := \bigwedge_{1 \le i \le n} \psi_i$$



Loops with Conditions



- How long may we **legally** stay at a cut q?
- **Intuition**: those $(\sigma_i, cons_i, Snd_i)$ are allowed to fire the self-loop (q, ψ, q) where
  - $cons_i \cup Snd_i$ comprises only irrelevant messages:
    - **weak mode**: no message from a direct successor cut is in,
    - **strict mode**: no message occurring in the LSC is in,
  - σ_i satisfies the local invariant relevant at q

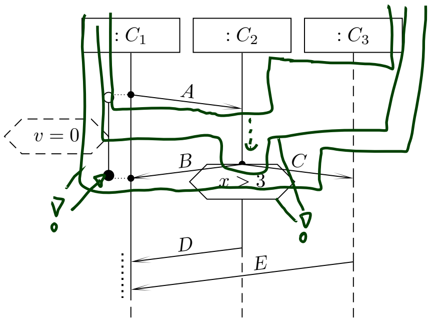
  And nothing else.
- **Formally**: Let $F := F_1 \cup \dots \cup F_n$ be the union of the firedsets of q.
  - $\psi := \underbrace{\neg(\bigvee \mathscr{E}(F))}_{=\mathit{true} \text{ if } F=\emptyset} \wedge \bigwedge \psi(q)$

Even More Helper Functions

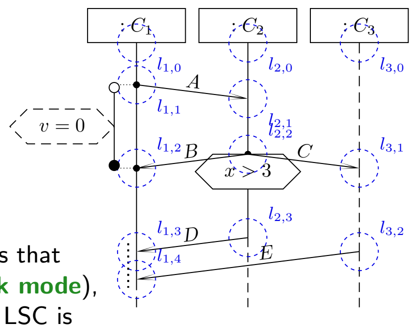
- **Constraints** relevant when moving from q **to** cut q':

$$\psi_\theta(q, q') = \{\psi \mid \exists L \subseteq \mathscr{L} \mid (L, \psi, \theta) \in \mathsf{Cond} \wedge L \cap (q' \setminus q) \neq \emptyset\}$$
$$\cup\, \psi_\theta(q')$$
$$\setminus \{\psi \mid \exists l \in q' \setminus q, l' \in \mathscr{L} \mid (l, \circ, expr, \theta, l') \in \mathsf{LocInv} \vee (l', expr, \theta, \circ, l) \in \mathsf{LocInv}\}$$
$$\cup \{\psi \mid \exists l \in q' \setminus q, l' \in \mathscr{L} \mid (l, \bullet, expr, \theta, l') \in \mathsf{LocInv} \vee (l', expr, \theta, \bullet, l) \in \mathsf{LocInv}\}$$

$$\psi(q, q') = \psi_{\mathsf{hot}}(q, q') \cup \psi_{\mathsf{cold}}(q, q')$$



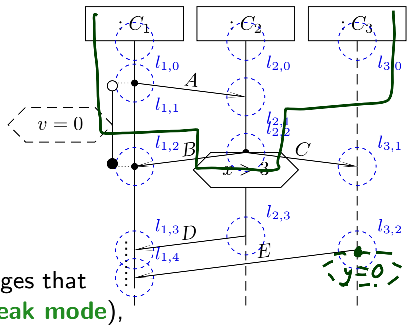
Progress with Conditions



- When do we move from q to q'?
- **Intuition**: those $(\sigma_i, cons_i, Snd_i)$ fire the progress transition (q, ψ, q') for which there exists a firedset F such that $q \rightsquigarrow_F q'$ and
  - $cons_i \cup Snd_i$ comprises exactly the messages that distinguish F from other firedsets of q (**weak mode**), and in addition no message occurring in the LSC is in $cons_i \cup Snd_i$ (**strict mode**),
  - σ_i satisfies the local inv. and condition relevant at q'
- **Formally**: Let $F, F_1, \dots, F_n$ be the firedsets of q and let $q \rightsquigarrow_F q'$ (unique).
  - $\psi := \bigwedge \mathscr{E}(F) \wedge \neg(\bigvee(\mathscr{E}(F_1) \cup \dots \cup \mathscr{E}(F_n)) \setminus \mathscr{E}(F)) \wedge \bigwedge \psi(q, q')$

Step III: Cold Conditions and Cold Local Invariants

Legal Exits



- When do we take a legal exit from q?
- **Intuition**: those $(\sigma_i, cons_i, Snd_i)$ fire the legal exit transition $(q, \psi, \mathscr{L})$
  - for which there exists a firedset F and some q' such that $q \rightsquigarrow_F q'$ and
    - $cons_i \cup Snd_i$ comprises exactly the messages that distinguish F from other firedsets of q (**weak mode**), and in addition no message occurring in the LSC is in $cons_i \cup Snd_i$ (**strict mode**) and
    - at least one cold condition or local invariant relevant when moving to q' is violated, or
  - for which there is no matching firedset and at least one cold local invariant relevant at q is violated.
- **Formally**: Let $F_1, \ldots, F_n$ be the firedsets of q with $q \rightsquigarrow_{F_i} q'_i$.
  - $\psi := \bigvee_{i=1}^{n} \bigwedge \mathscr{E}(F_i) \wedge \neg(\bigvee(\mathscr{E}(F_1) \cup \cdots \cup \mathscr{E}(F_n)) \setminus \mathscr{E}(F_i)) \wedge \bigvee \psi_{\mathsf{cold}}(q, q'_i)$
    $\vee\ \neg(\bigvee \mathscr{E}(F_i)) \wedge \bigvee \psi_{\mathsf{cold}}(q)$

Finally: The LSC Semantics

A **full LSC** L consist of

- a **body** $(I, (\mathscr{L}, \preceq), \sim, \mathscr{S}, \mathsf{Msg}, \mathsf{Cond}, \mathsf{LocInv})$,
- an **activation condition** (here: event) $ac = E^{?}_{i_1,i_2}$, $E \in \mathscr{E}$, $i_1, i_2 \in I$,
- an **activation mode**, either **initial** or **invariant**,
- a **chart mode**, either **existential** (cold) or **universal** (hot).

A set W of words over $\mathscr{S}$ and $\mathscr{D}$ **satisfies** L, denoted $W \models L$, iff L

- **universal** (= hot), **initial**, and

$$\forall\, w \in W\ \forall\, \beta : I \to \mathrm{dom}(\sigma(w^0)) \bullet w \text{ activates } L \implies w \in \mathcal{L}_\beta(\mathcal{B}_L).$$

- **existential** (= cold), **initial**, and

$$\exists\, w \in W\ \exists\, \beta : I \to \mathrm{dom}(\sigma(w^0)) \bullet w \text{ activates } L \wedge w \in \mathcal{L}_\beta(\mathcal{B}_L).$$

- **universal** (= hot), **invariant**, and

suffix starting at k

$$\forall\, w \in W\ \forall\, k \in \mathbb{N}_0\ \forall\, \beta : I \to \mathrm{dom}(\sigma(w^k)) \bullet w/k \text{ activates } L \implies w/k \in \mathcal{L}_\beta(\mathcal{B}_L).$$

- **existential** (= cold), **invariant**, and

$$\exists\, w \in W\ \exists\, k \in \mathbb{N}_0\ \exists\, \beta : I \to \mathrm{dom}(\sigma(w^k)) \bullet w/k \text{ activates } L \wedge w/k \in \mathcal{L}_\beta(\mathcal{B}_L).$$

Back to UML: Interactions

Model Consistency wrt. Interaction

- We assume that the set of interactions $\mathscr{I}$ is partitioned into two (possibly empty) sets of **universal** and **existential** interactions, i.e.

$$\mathscr{I} = \mathscr{I}_\forall \,\dot\cup\, \mathscr{I}_\exists.$$

Definition. A model

$$\mathcal{M} = (\mathscr{CD}, \mathscr{SM}, \mathscr{OD}, \mathscr{I})$$

is called **consistent** (more precise: the constructive description of behaviour is consistent with the reflective one) if and only if

$$\forall \mathcal{I} \in \mathscr{I}_\forall : \mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{I})$$

and

$$\forall \mathcal{I} \in \mathscr{I}_\exists : \mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\mathcal{I}) \neq \emptyset.$$

Interactions as Reflective Description

- In UML, reflective (temporal) descriptions are subsumed by **interactions**.
- A UML model $\mathcal{M} = (\mathscr{CD}, \mathscr{SM}, \mathscr{OD}, \mathscr{I})$ has a set of interactions $\mathscr{I}$.
- An interaction $\mathcal{I} \in \mathscr{I}$ can be (OMG claim: equivalently) **diagrammed** as
  - **sequence diagram**, **timing diagram**, or
  - **communication diagram** (formerly known as collaboration diagram).

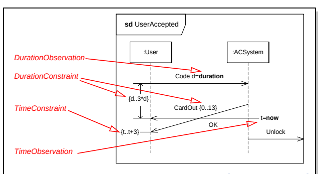


Figure 14.26 - Sequence Diagram with time and timing concepts [OMG, 2007b, 513]

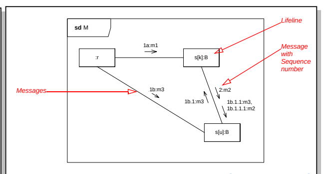


Figure 14.27 - Communication diagram [OMG, 2007b, 515]

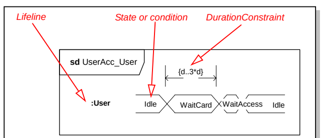


Figure 14.30 - Compact Lifeline with States [OMG, 2007b, 522]

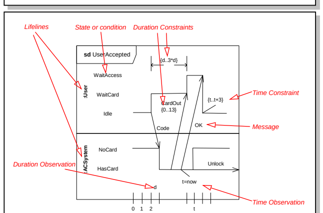
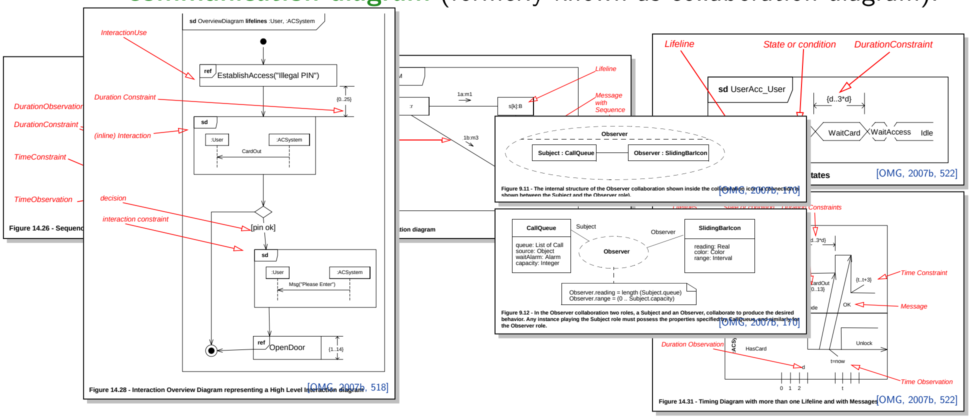


Figure 14.31 - Timing Diagram with more than one Lifeline and with Messages [OMG, 2007b, 522]

Interactions as Reflective Description

- In UML, reflective (temporal) descriptions are subsumed by **interactions**.
- A UML model $\mathcal{M} = (\mathscr{C}\mathscr{D}, \mathscr{S}\mathscr{M}, \mathscr{O}\mathscr{D}, \mathscr{I})$ has a set of interactions $\mathscr{I}$.
- An interaction $\mathcal{I} \in \mathscr{I}$ can be (OMG claim: equivalently) **diagrammed** as
  - **sequence diagram**, **timing diagram**, or
  - **communication diagram** (formerly known as collaboration diagram).



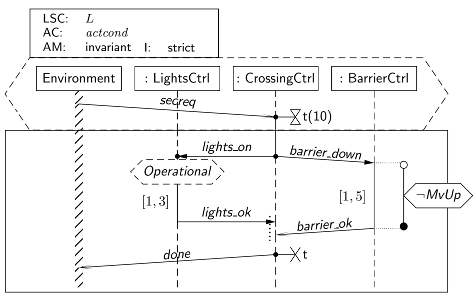
Why Sequence Diagrams?

Most Prominent: Sequence Diagrams — with **long history**:

- **Message Sequence Charts**, standardized by the ITU in different versions, often accused to lack a formal semantics.
- **Sequence Diagrams** of UML 1.x

Most severe **drawbacks** of these formalisms:

- unclear **interpretation**: example scenario or invariant?
- unclear **activation**: what triggers the requirement?
- unclear **progress** requirement: must all messages be observed?
- **conditions** merely comments
- no means to express **forbidden scenarios**



Thus: Live Sequence Charts

- **SDs of UML 2.x** address **some** issues, yet the standard exhibits unclarities and even contradictions [Harel and Maoz, 2007, Störrle, 2003]
- For the lecture, we consider **Live Sequence Charts** (LSCs) [Damm and Harel, 2001, Klose, 2003, Harel and Marelly, 2003], who have a common fragment with UML 2.x SDs [Harel and Maoz, 2007]
- **Modelling guideline**: stick to that fragment.

Side Note: Protocol Statemachines

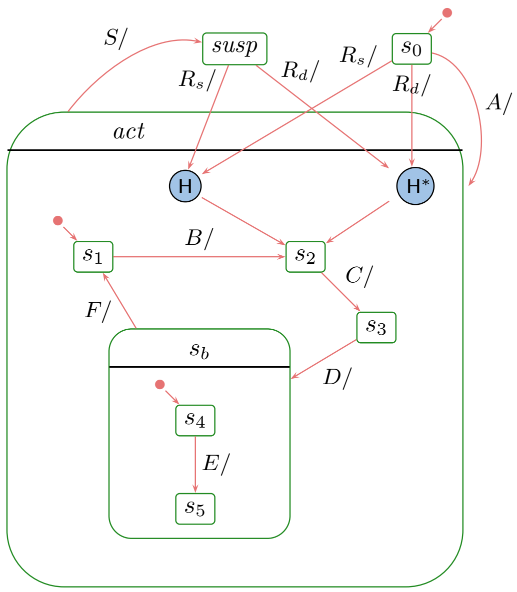
Same direction: **call orders** on operations

- **"for each C instance, method $f()$ shall only be called after $g()$ but before $h()$"**

Can be formalised with protocol state machines.

The Concept of History, and Other Pseudo-States

History and Deep History: By Example



What happens on...

- R_s?

  s_0, s_2
- R_d?

  s_0, s_2
- A, B, C, S, R_s?

  $s_0, s_1, s_2, s_3, susp, s_3$
- A, B, S, R_d?

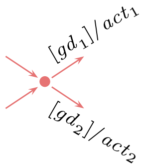
  $s_0, s_1, s_2, s_3, susp, s_3$
- A, B, C, D, E, R_s?

  $s_0, s_1, s_2, s_3, s_4, s_5, susp, s_3$
- A, B, C, D, R_d?

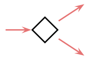
  $s_0, s_1, s_2, s_3, s_4, s_5, susp, s_5$

Junction and Choice

- Junction (**"static conditional branch"**):



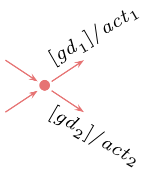
- Choice: (**"dynamic conditional branch"**)



Note: not so sure about naming and symbols, e.g.,
I'd guessed it was just the other way round...

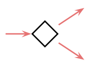
Junction and Choice

- Junction (**"static conditional branch"**):



  - **good**: abbreviation
  - unfolds to so many similar transitions with different guards,
    the unfolded transitions are then checked for enabledness
  - at best, start with trigger, branch into conditions, then apply actions

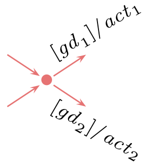
- Choice: (**"dynamic conditional branch"**)



Note: not so sure about naming and symbols, e.g.,
I'd guessed it was just the other way round...

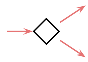
Junction and Choice

- Junction (**"static conditional branch"**):



  - **good**: abbreviation
  - unfolds to so many similar transitions with different guards, the unfolded transitions are then checked for enabledness
  - at best, start with trigger, branch into conditions, then apply actions

- Choice: (**"dynamic conditional branch"**)

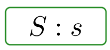


  - **evil**: may get stuck
  - enters the transition **without knowing** whether there's an enabled path
  - at best, use "else" and convince yourself that it cannot get stuck
  - maybe even better: **avoid**

Note: not so sure about naming and symbols, e.g.,
I'd guessed it was just the other way round...

Entry and Exit Point, Submachine State, Terminate

- Hierarchical states can be **"folded"** for readability.
  (but: this can also hinder readability.)
- Can even be taken from a different state-machine for re-use.



Entry and Exit Point, Submachine State, Terminate

- Hierarchical states can be **“folded”** for readability.
  (but: this can also hinder readability.)
- Can even be taken from a different state-machine for re-use. $\boxed{S : s}$
- **Entry/exit points** $\bigcirc$, $\otimes$
  - Provide connection points for finer integration into the current level, than just via initial state.
  - Semantically a bit tricky:
    - **First** the exit action of the exiting state,
    - **then** the actions of the transition,
    - **then** the entry actions of the entered state,
    - **then** action of the transition from the entry point to an internal state,
    - and **then** that internal state’s entry action.

Entry and Exit Point, Submachine State, Terminate

- Hierarchical states can be **“folded”** for readability.
  (but: this can also hinder readability.)
- Can even be taken from a different state-machine for re-use. $\boxed{S : s}$
- **Entry/exit points** $\bigcirc$, $\otimes$
  - Provide connection points for finer integration into the current level, than just via initial state.
  - Semantically a bit tricky:
    - **First** the exit action of the exiting state,
    - **then** the actions of the transition,
    - **then** the entry actions of the entered state,
    - **then** action of the transition from the entry point to an internal state,
    - and **then** that internal state’s entry action.

- **Terminate Pseudo-State** $\times$
  - When a terminate pseudo-state is reached, the object taking the transition is immediately killed.

Deferred Events in State-Machines

Deferred Events: Idea

For ages, UML state machines comprises the feature of **deferred events**.

The idea is as follows:

- Consider the following state machine:

$$\boxed{s_1} \xrightarrow{E/} \boxed{s_2} \xrightarrow{F/} \boxed{s_3}$$

- Assume we're stable in s_1, and F is ready in the ether.
- In **the framework of the course**, F is **discarded**.

Deferred Events: Idea

For ages, UML state machines comprises the feature of **deferred events**.

The idea is as follows:

- Consider the following state machine:

$$\boxed{s_1} \xrightarrow{E/} \boxed{s_2} \xrightarrow{F/} \boxed{s_3}$$

- Assume we're stable in s_1, and F is ready in the ether.
- In **the framework of the course**, F is **discarded**.
- But we **may** find it a pity to discard the poor event and **may** want to remember it for later processing, e.g. in s_2, in other words, **defer** it.

Deferred Events: Idea

For ages, UML state machines comprises the feature of **deferred events**.

The idea is as follows:

- Consider the following state machine:

$$\boxed{s_1} \xrightarrow{E/} \boxed{s_2} \xrightarrow{F/} \boxed{s_3}$$

- Assume we're stable in s_1, and F is ready in the ether.
- In **the framework of the course**, F is **discarded**.
- But we **may** find it a pity to discard the poor event and **may** want to remember it for later processing, e.g. in s_2, in other words, **defer** it.

General options to satisfy such needs:

- Provide a pattern how to “program” this (use self-loops and helper attributes).
- Turn it into an original language concept.

Deferred Events: Idea

For ages, UML state machines comprises the feature of **deferred events**.

The idea is as follows:

- Consider the following state machine:

$$s_1 \xrightarrow{E/} s_2 \xrightarrow{F/} s_3$$

- Assume we're stable in s_1, and F is ready in the ether.
- In **the framework of the course**, F is **discarded**.
- But we **may** find it a pity to discard the poor event and **may** want to remember it for later processing, e.g. in s_2, in other words, **defer** it.

General options to satisfy such needs:

- Provide a pattern how to “program” this (use self-loops and helper attributes).
- Turn it into an original language concept. **(← OMG's choice)**

Deferred Events: Syntax and Semantics

- **Syntactically**,
  - Each state has (in addition to the name) a set of deferred events.
  - **Default**: the empty set.

Deferred Events: Syntax and Semantics

- **Syntactically**,
  - Each state has (in addition to the name) a set of deferred events.
  - **Default**: the empty set.
- The **semantics** is a bit intricate, something like
  - if an event E is dispatched,
  - and there is no transition enabled to consume E,
  - and E is in the deferred set of the current state configuration,
  - then stuff E into some "deferred events space" of the object, (e.g. into the ether ($=$ extend ε) or into the local state of the object ($=$ extend σ))
  - and turn attention to the next event.

– 19 – 2013-01-23 – Sdefer –

Deferred Events: Syntax and Semantics

- **Syntactically**,
  - Each state has (in addition to the name) a set of deferred events.
  - **Default**: the empty set.
- The **semantics** is a bit intricate, something like
  - if an event E is dispatched,
  - and there is no transition enabled to consume E,
  - and E is in the deferred set of the current state configuration,
  - then stuff E into some "deferred events space" of the object, (e.g. into the ether ($=$ extend ε) or into the local state of the object ($=$ extend σ))
  - and turn attention to the next event.

- **Not so obvious**:
  - Is there a priority between deferred and regular events?
  - Is the order of deferred events preserved?
  - ...

  [Fecher and Schönborn, 2007], e.g., claim to provide semantics for the complete Hierarchical State Machine language, including deferred events.

– 19 – 2013-01-23 – Sdefer –

Active and Passive Objects [Harel and Gery, 1997]

What about non-Active Objects?

Recall:

- We're **still** working under the assumption that all classes in the class diagram (and thus all objects) are **active**.
- That is, each object has its own thread of control and is (if stable) at any time ready to process an event from the ether.

What about non-Active Objects?

Recall:

- We're **still** working under the assumption that all classes in the class diagram (and thus all objects) are **active**.
- That is, each object has its own thread of control and is (if stable) at any time ready to process an event from the ether.

But the world doesn't consist of only active objects.

For instance, in the crossing controller from the exercises we could wish to have the whole system live in one thread of control.

So we have to address questions like:

- Can we send events to a non-active object?
- And if so, when are these events processed?
- etc.

Active and Passive Objects: Nomenclature

[Harel and Gery, 1997] propose the following (orthogonal!) notions:

- A class (and thus the instances of this class) is either **active** or **passive** as declared in the class diagram.
  - An **active** object has (in the operating system sense) an own thread: an own program counter, an own stack, etc.
  - A **passive** object doesn't.

Active and Passive Objects: Nomenclature

[Harel and Gery, 1997] propose the following (orthogonal!) notions:

- A class (and thus the instances of this class) is either **active** or **passive** as declared in the class diagram.
  - An **active** object has (in the operating system sense) an own thread: an own program counter, an own stack, etc.
  - A **passive** object doesn't.
- A class is either **reactive** or **non-reactive**.
  - A **reactive** class has a (non-trivial) state machine.
  - A **non-reactive** one hasn't.

Active and Passive Objects: Nomenclature

[Harel and Gery, 1997] propose the following (orthogonal!) notions:

- A class (and thus the instances of this class) is either **active** or **passive** as declared in the class diagram.
  - An **active** object has (in the operating system sense) an own thread: an own program counter, an own stack, etc.
  - A **passive** object doesn't.
- A class is either **reactive** or **non-reactive**.
  - A **reactive** class has a (non-trivial) state machine.
  - A **non-reactive** one hasn't.

Which combinations do we understand?

	active	passive
reactive	✔	(*)
non-reactive	(✔)	(✔)

Passive and Reactive

- So why don't we understand passive/reactive?
- Assume passive objects u_1 and u_2, and active object u, and that there are events in the ether for all three.

  Which of them (can) start a run-to-completion step...?
  Do run-to-completion steps still interleave...?

– 19 – 2013-01-23 – Sactpass –

Passive and Reactive

- So why don't we understand passive/reactive?
- Assume passive objects u_1 and u_2, and active object u, and that there are events in the ether for all three.

  Which of them (can) start a run-to-completion step...?
  Do run-to-completion steps still interleave...?

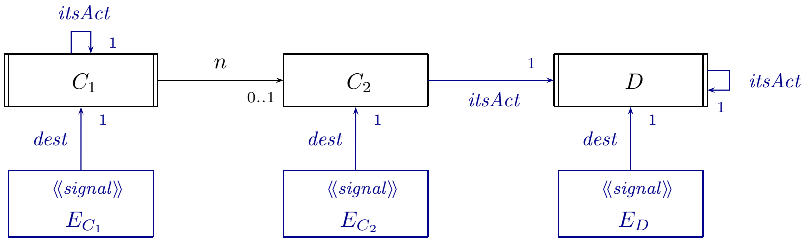
Reasonable Approaches:

- **Avoid** — for instance, by
  - require that **reactive implies active** for model well-formedness.
  - requiring for model well-formedness that events are **never sent** to instances of non-reactive classes.
- **Explain** — here: (following [Harel and Gery, 1997])
  - Delegate all dispatching of events to the active objects.

– 19 – 2013-01-23 – Sactpass –

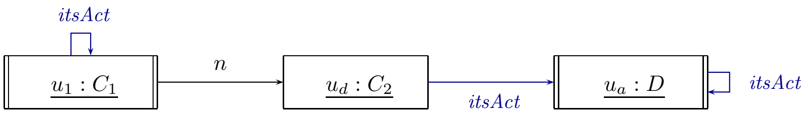
Passive Reactive Classes

- Firstly, establish that each object u knows, via (implicit) link $itsAct$, **the active object** u_{act} which is responsible for dispatching events to u.
- If u is an instance of an active class, then $u_a = u$.



Passive Reactive Classes

- Firstly, establish that each object u knows, via (implicit) link $itsAct$, **the active object** u_{act} which is responsible for dispatching events to u.
- If u is an instance of an active class, then $u_a = u$.

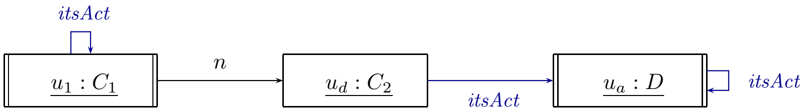


Sending an event:

- Establish that of each signal we have a version E_C with an association $dest : C_{0,1}$, $C \in \mathscr{C}$.
- Then $n!E$ in $u_1 : C_1$ becomes:
- Create an instance u_e of E_{C_2} and set u_e's $dest$ to $u_d := \sigma(u_1)(n)$.
- Send to $u_a := \sigma(\sigma(u_1)(n))(itsAct)$, i.e., $\varepsilon' = \varepsilon \oplus (u_a, u_e)$.

Passive Reactive Classes

- Firstly, establish that each object u knows, via (implicit) link $itsAct$, **the active object** u_{act} which is responsible for dispatching events to u.
- If u is an instance of an active class, then $u_a = u$.



Sending an event:

- Establish that of each signal we have a version E_C with an association $dest : C_{0,1}$, $C \in \mathscr{C}$.
- Then $n!E$ in $u_1 : C_1$ becomes:
- Create an instance u_e of E_{C_2} and set u_e's $dest$ to $u_d := \sigma(u_1)(n)$.
- Send to $u_a := \sigma(\sigma(u_1)(n))(itsAct)$, i.e., $\varepsilon' = \varepsilon \oplus (u_a, u_e)$.

Dispatching an event:

- Observation: the ether only has events for active objects.
- Say u_e is ready in the ether for u_a.
- Then u_a asks $\sigma(u_e)(dest) = u_d$ to process u_e — and waits until completion of corresponding RTC.
- u_d may in particular discard event.

And What About Methods?

And What About Methods?

- In the current setting, the (local) state of objects is only modified by actions of transitions, which we abstract to transformers.
- In general, there are also **methods**.
- UML follows an approach to separate
  - the **interface declaration** from
  - the **implementation**.

  In C++ lingo: distinguish **declaration** and **definition** of method.

And What About Methods?

- In the current setting, the (local) state of objects is only modified by actions of transitions, which we abstract to transformers.
- In general, there are also **methods**.
- UML follows an approach to separate
  - the **interface declaration** from
  - the **implementation**.

  In C++ lingo: distinguish **declaration** and **definition** of method.

- In UML, the former is called **behavioural feature** and can (roughly) be
  - a **call interface** $f(\tau_{1_1}, \dots, \tau_{n_1}) : \tau_1$
  - a **signal name** E

C
$\xi_1\ f(\tau_{1,1}, \dots, \tau_{1,n_1}) : \tau_1\ P_1$ $\xi_2\ F(\tau_{2,1}, \dots, \tau_{2,n_2}) : \tau_2\ P_2$
$\langle\langle signal \rangle\rangle\ E$

Note: The signal list can be seen as redundant (can be looked up in the state machine) of the class. But: certainly useful for documentation (or sanity check).

Behavioural Features

C
$\xi_1\ f(\tau_{1,1}, \dots, \tau_{1,n_1}) : \tau_1\ P_1$ $\xi_2\ F(\tau_{2,1}, \dots, \tau_{2,n_2}) : \tau_2\ P_2$
$\langle\langle signal \rangle\rangle\ E$

Semantics:

- The **implementation** of a behavioural feature can be provided by:
  - An **operation**.
  - The class' **state-machine** ("triggered operation").

Behavioural Features

C
$\xi_1\ f(\tau_{1,1}, \dots, \tau_{1,n_1}) : \tau_1\ P_1$ $\xi_2\ F(\tau_{2,1}, \dots, \tau_{2,n_2}) : \tau_2\ P_2$
$\langle\langle signal \rangle\rangle\ E$

Semantics:

- The **implementation** of a behavioural feature can be provided by:
  - An **operation**.

    In our setting, we simply assume a transformer like T_f.

    It is then, e.g. clear how to admit method calls as actions on transitions: function composition of transformers (clear but tedious: non-termination).

    In a setting with Java as action language: operation is a method body.
  - The class' **state-machine** ("triggered operation").

Behavioural Features

C
$\xi_1\ f(\tau_{1,1}, \dots, \tau_{1,n_1}) : \tau_1\ P_1$ $\xi_2\ F(\tau_{2,1}, \dots, \tau_{2,n_2}) : \tau_2\ P_2$
$\langle\langle signal \rangle\rangle\ E$

Semantics:

- The **implementation** of a behavioural feature can be provided by:
  - An **operation**.

    In our setting, we simply assume a transformer like T_f.

    It is then, e.g. clear how to admit method calls as actions on transitions: function composition of transformers (clear but tedious: non-termination).

    In a setting with Java as action language: operation is a method body.
  - The class' **state-machine** ("triggered operation").
    - Calling F with n_2 parameters for a stable instance of C creates an auxiliary event F and dispatches it (bypassing the ether).
    - Transition actions may fill in the return value.
    - On completion of the RTC step, the call returns.
    - For a non-stable instance, the caller blocks until stability is reached again.

Behavioural Features: Visibility and Properties

C
$\xi_1\ f(\tau_{1,1}, \dots, \tau_{1,n_1}) : \tau_1\ P_1$ $\xi_2\ F(\tau_{2,1}, \dots, \tau_{2,n_2}) : \tau_2\ P_2$
$\langle\langle signal \rangle\rangle\ E$

- **Visibility**:
  - Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods.

Behavioural Features: Visibility and Properties

C
$\xi_1\ f(\tau_{1,1},\dots,\tau_{1,n_1}) : \tau_1\ P_1$ $\xi_2\ F(\tau_{2,1},\dots,\tau_{2,n_2}) : \tau_2\ P_2$
$\langle\langle signal\rangle\rangle\ E$

- **Visibility**:
  - Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods.
- **Useful properties**:
  - **concurrency**
    - **concurrent** — is thread safe
    - **guarded** — some mechanism ensures/should ensure mutual exclusion
    - **sequential** — is not thread safe, users have to ensure mutual exclusion
  - **isQuery** — doesn't modify the state space (thus thread safe)
- For simplicity, we leave the notion of steps untouched, we construct our semantics around state machines.
  Yet we could explain pre/post in OCL (if we wanted to).

References

References

[Damm and Harel, 2001] Damm, W. and Harel, D. (2001). LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80.

[Fecher and Schönborn, 2007] Fecher, H. and Schönborn, J. (2007). UML 2.0 state machines: Complete formal semantics via core state machines. In Brim, L., Haverkort, B. R., Leucker, M., and van de Pol, J., editors, *FMICS/PDMC*, volume 4346 of *LNCS*, pages 244–260. Springer.

[Harel and Gery, 1997] Harel, D. and Gery, E. (1997). Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42.

[Harel and Maoz, 2007] Harel, D. and Maoz, S. (2007). Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling (SoSyM)*. To appear. (Early version in SCESM'06, 2006, pp. 13-20).

[Harel and Marelly, 2003] Harel, D. and Marelly, R. (2003). *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag.

[Klose, 2003] Klose, J. (2003). *LSCs: A Graphical Formalism for the Specification of Communication Behavior*. PhD thesis, Carl von Ossietzky Universität Oldenburg.

[OMG, 2007a] OMG (2007a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-04.

[OMG, 2007b] OMG (2007b). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.

[Störrle, 2003] Störrle, H. (2003). Assert, negate and refinement in UML-2 interactions. In Jürjens, J., Rumpe, B., France, R., and Fernandez, E. B., editors, *CSDUML 2003*, number TUM-I0323. Technische Universität München.