Lecture 10: Live Sequence Charts & RE Wrap-Up

2018-06-11

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Requirements Engineering: Content

- Introduction
- Requirements Specification
  - Desired Properties
  - Kinds of Requirements
  - Analysis Techniques
- Documents
  - Dictionary, Specification
- Specification Languages
  - Natural Language
  - Decision Tables
    - Syntax, Semantics
    - Completeness, Consistency, ...
  - Scenarios
    - User Stories, Use Cases
    - Live Sequence Charts
      - Syntax, Semantics
- Definition: Software & SW Specification
- Wrap-Up
LSC Semantics: TBA Construction
The TBA $B(L)$ of LSC $L$ over $C$ and $E$ is $(G_B, Q, q_{\text{init}}, \rightarrow, Q_F)$ with

- $C_B = C \cup E_I^T$, where $E_I^T = \{ E_{i,j}^T \mid E \in E, i,j \in \mathbb{I} \}$.
- $Q$ is the set of cuts of $L$, $q_{\text{init}}$ is the instance heads cut.
- $\rightarrow$ consists of loops, progress transitions (from $\rightarrow\gamma$), and legal exits (cold cond./local inv.).
- $Q_F = \{ C \in Q \mid \Theta(C) = \text{cold} \lor C = C \}$ is the set of cold cuts and the maximal cut.
Recall: The TBA $\mathcal{B}(L)$ of LSC $L$ is $(C, Q, q_{ini}, \rightarrow, Q_F)$ with

- $Q$ is the set of cuts of $L$, $q_{ini}$ is the instance heads cut,
- $C_H = C \cup E'$,
- $\rightarrow$ consists of loops, progress transitions (from $\rightarrow F$), and legal exits (cold cond./local inv.),
- $Q_F = \{ C \in Q \mid \Theta(C) = \text{cold} \lor C = L \}$ is the set of cold cuts.

So in the following, we “only” need to construct the transitions’ labels:

$$\rightarrow = \{(q, \psi_{\text{loop}}(q)) \mid q \in Q\} \cup \{(q, \psi_{\text{prog}}(q, q')) \mid q \rightarrow_F q'\} \cup \{(q, \psi_{\text{exit}}(q), L) \mid q \in Q\}$$
**Loop Condition**

\[
\psi_{\text{loop}}(q) = \psi_{\text{Msg}}(q) \land \psi_{\text{LocInv}}(q) \land \psi_{\text{Cond}}(q)
\]

- \(\psi_{\text{Msg}}(q) = \neg \bigvee_{1 \leq i \leq n} \psi_{\text{Msg}}(q, q_i) \land (\text{strict} \implies \psi_{\text{Msg}}(q, q_i) \land \psi_{\text{LocInv}}(q, q_i) \land \psi_{\text{LocInv}}(q, q_j))
\]

- \(\psi_{\text{LocInv}}(q) = \bigwedge_{l=(L,\theta), \lambda \in \text{LocInv}, \Theta(l) = \theta, l \text{ active at } q} \psi_{\text{LocInv}}(q, l)
\]

A location \(l\) is called **front location** of cut \(C\) if and only if \(\nexists \ l' \in C \mid l < l'\).

Local invariant \((l_0, \iota_0, \phi, l_1, \iota_1)\) is active at cut \((l)\) if and only if \(l_0 \leq l < l_1\) for some front location \(l\) of cut \(q\) or \(l = l_1 \land \iota_1 = \star\).

- \(\text{Msg}(F) = \{E_{l}(l', l') | (L, E, l) \in \text{Msg}, l \in F\} \cup \{E_{l}(l, l') | (L, E, l') \in \text{Msg}, l' \in F\}
\]

- \(\text{Msg}(F_1, \ldots, F_n) = \bigcup_{1 \leq i \leq n} \text{Msg}(F_i)
\]

**Progress Condition**

\[
\psi_{\text{Cond}}(q, q_1) = \psi_{\text{Msg}}(q, q_1) \land \psi_{\text{LocInv}}(q, q_1) \land \psi_{\text{Cond}}(q, q_1) \land \psi_{\text{LocInv}}(q, q_1)
\]

- \(\psi_{\text{Cond}}(q, q_1) = \bigwedge_{l=(L, \theta), \lambda \in \text{Cond}, \Theta(l) = \theta, L'(q_1 \backslash q) \neq \emptyset} \psi_{\text{Cond}}(q, q_1)
\]

- \(\psi_{\text{LocInv}}(q, l_1, q_1) = \bigwedge_{l=(L, \theta), \lambda \in \text{LocInv}, \Theta(l) = \theta, \lambda \text{ active at } q} \psi_{\text{LocInv}}(q, l_1, q_1)
\]

Local invariant \((l_0, \iota_0, \phi, l_1, \iota_1)\) is active at \(q\) if and only if
- \(l_0 < l < l_1\), or
- \(l = l_0 \land \iota_0 = \star\), or
- \(l = l_1 \land \iota_1 = \star\)

for some front location \(l\) of cut \((l)\).
Example (without strictness condition)
Content

- Live Sequence Charts
  - TBA Construction
  - LSCs vs. Software
    - Software and Software Specification, formally
    - Software satisfies Software Specification
  - Full LSC (without pre-chart)
    - Activation Condition & Activation Mode
  - (Slightly) Advanced LSC Topics
    - Full LSC with pre-chart
  - LSCs in Requirements Engineering
    - strengthening existential LSCs (scenarios) into universal LSCs (requirements)
  - LSCs in Quality Assurance

- Requirements Engineering Wrap-Up
  - Requirements Analysis in a Nutshell
  - Recall: Validation by Translation

- Outlook: Formal Methods in Design & QA

Excursion: Symbolic Büchi Automata
Symbolic Büchi Automata

**Definition.** A **Symbolic Büchi Automaton** (TBA) is a tuple

\[ B = (C_B, Q, q_{init}, \rightarrow, Q_F) \]

where

- \( C_B \) is a set of atomic propositions,
- \( Q \) is a finite set of states,
- \( q_{init} \in Q \) is the initial state,
- \( \rightarrow \subseteq Q \times \Phi(C_B) \times Q \) is the finite transition relation. Each transition \( (q, \psi, q') \in \rightarrow \) from state \( q \) to state \( q' \) is labelled with a propositional formula \( \psi \in \Phi(C_B) \).
- \( Q_F \subseteq Q \) is the set of fair (or accepting) states.

**Example:**

\[ B_{sym}: \Sigma = \{a, b, c, d\} \rightarrow B \]
**Definition.** Let \( B = (C_B, Q, q_{ini}, \rightarrow, Q_F) \) be a TBA and
\[
w = \sigma_1, \sigma_2, \sigma_3, \ldots \in (C_B \rightarrow B)^\omega
\]
an infinite word, each letter is a valuation of \( C_B \).
An infinite sequence
\[
\varrho = q_0, q_1, q_2, \ldots \in Q^\omega
\]
of states is called run of \( B \) over \( w \) if and only if
1. \( q_0 = q_{ini} \),
2. for each \( i \in \mathbb{N}_0 \) there is a transition \( (q_i, \psi_i, q_{i+1}) \in \rightarrow \text{ s.t. } \sigma_i \models \psi_i \).

**Example:**
\[
\begin{array}{c}
\text{B}_{sym}: \\
\begin{array}{c}
\text{Σ = } \{(a, b, c, d) \rightarrow B\}
\end{array}
\end{array}
\]
\[
w = \{a \mapsto \text{true}, b \mapsto \text{true}, c \mapsto \text{false}, d \mapsto \text{false}\}, \{c\}, \{a, b\}, (\{d\}, \{a, b\})^\omega
\]
\[
\{a, b\} \text{ for short}
\]

**The Language of a TBA**

**Definition.** We say TBA \( B = (C_B, Q, q_{ini}, \rightarrow, Q_F) \) accepts the word
\[
w = (\sigma_i)_{i \in \mathbb{N}_0} \in (C_B \rightarrow B)^\omega
\]
if and only if \( B \) has a run \( \varrho \) of states that are visited infinitely often by \( \varrho \), i.e.,
\[
\forall i \in \mathbb{N}_0 \exists j > i : q_j \in Q_F.
\]
We call the set \( \text{Lang}(B) \subseteq (C_B \rightarrow B)^\omega \) of words that are accepted by \( B \) the language of \( B \).
$w = \{\}, \{E_{1}^{U,V}, E_{1}^{U,V}\}, \{pSOFT_{1}^{U,V}, pSOFT_{1}^{U,V}\}, \{\}, \{\}, \{SOFT_{1}^{V,U}, SOFT_{1}^{V,U}\}, \{\} \ldots \\
\in \text{Lang}(\mathcal{B}(\mathcal{L}))$
Excursion: Software Specification, Formally

**Formal Methods in Requirements Engineering**

- Recall:
  - We would like to precisely and objectively specify the set of allowed softwares that make the customer happy.
  - (The designers and developers then choose one from the set.)

- In other words, we want to formally define a satisfies relation between softwares and software specifications.
  - That is, given a software \( S \) and a software specification \( \mathcal{F} \), we want to define when (and only when) software \( S \) satisfies software specification \( \mathcal{F} \), denoted by
    \[
    S \models \mathcal{F}.
    \]

- Once again:
  - \( S \models \mathcal{F} \): specification is satisfied, \( S \) is one "allowed" design, should be accepted.
  - \( S \not\models \mathcal{F} \): specification is not satisfied, \( S \) may not satisfy customer's needs.
Software, formally

**Definition.** Software is a finite description $S$ of a (possibly infinite) set $[S]$ of (finite or infinite) computation paths of the form

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \ldots$$

where

- $\sigma_i \in \Sigma, i \in \mathbb{N}_0$, is called state (or configuration), and
- $\alpha_i \in A, i \in \mathbb{N}_0$, is called action (or event).

The (possibly partial) function $[\cdot] : S \mapsto [S]$ is called interpretation of $S$.

**Example:** room ventilation system

- $\Sigma = \{b, on, off, go, stop\} \rightarrow \mathbb{B}$, $A = \{\tau\}$.
- computation path:

  \[
  \begin{pmatrix}
  b \mapsto false \\
  on \mapsto false \\
  off \mapsto true \\
  go \mapsto false \\
  stop \mapsto false \\
  \end{pmatrix} =
  \begin{pmatrix}
  \text{off } \\
  \text{on } \\
  \end{pmatrix}
  \xrightarrow{\tau}
  \begin{pmatrix}
  b \\
  off \\
  \end{pmatrix}
  \xrightarrow{\tau}
  \begin{pmatrix}
  \text{on } \\
  \text{off } \\
  \end{pmatrix}
  \xrightarrow{\tau}
  \begin{pmatrix}
  b \\
  \text{stop } \\
  \end{pmatrix}
  \xrightarrow{\tau}
  \begin{pmatrix}
  \text{off } \\
  \end{pmatrix}
  \ldots
  \]

Software, formally

**Definition.** Software is a finite description $S$ of a (possibly infinite) set $[S]$ of (finite or infinite) computation paths of the form

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \ldots$$

where

- $\sigma_i \in \Sigma, i \in \mathbb{N}_0$, is called state (or configuration), and
- $\alpha_i \in A, i \in \mathbb{N}_0$, is called action (or event).

The (possibly partial) function $[\cdot] : S \mapsto [S]$ is called interpretation of $S$.

**Example:** vending machine

- computation path:

  \[
  \sigma_0 \xrightarrow{\tau} \sigma_1 \xrightarrow{E_{1,U,V}} \sigma_2 \xrightarrow{psOFT_{U,V}} \sigma_3 \xrightarrow{\tau} \sigma_4 \xrightarrow{\tau} \sigma_5 \xrightarrow{\tau} \sigma_6 \xrightarrow{SOFT_{V,U}} \ldots
  \]

  - machine switched on
  - user inserts $f \in \text{buttons}$ light up
  - user presses SOFT button
  - prepare dispenser
  - drink ready
  - notify user
Definition. A software specification $\mathcal{S}$ is a finite description of a (possibly infinite) set $\mathcal{L}$ of softwares, i.e.

$$\mathcal{L} = \{(S_1, [\cdot]), (S_2, [\cdot]), \ldots \}.$$ The (possibly partial) function $[\cdot] : \mathcal{S} \mapsto \mathcal{L}$ is called interpretation of $\mathcal{S}$.

Definition. Software $(S, [\cdot])$ satisfies software specification $\mathcal{S}$, denoted by $S \models \mathcal{S}$, if and only if

$$(S, [\cdot]) \in \mathcal{L}.$$ 

Software Satisfies Software Specification: Example

Software Specification $\mathcal{S}$:

<table>
<thead>
<tr>
<th>$T$: room ventilation</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$  button pressed?</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$-$</td>
</tr>
<tr>
<td>$off$  ventilation off?</td>
<td>$-$</td>
<td>$-$</td>
<td>$+$</td>
</tr>
<tr>
<td>$on$  ventilation on?</td>
<td>$-$</td>
<td>$-$</td>
<td>$+$</td>
</tr>
<tr>
<td>$go$ start ventilation</td>
<td>$\times$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>stop stop ventilation</td>
<td>$-$</td>
<td>$\times$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

Define: $(S, [\cdot]) \in \mathcal{L}$ if and only if for all

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \in [S]$$

and for all $i \in \mathbb{N}_0$,

$$\exists \tau \in T \bullet r_i \models \mathcal{F}(r_i).$$

Software:

- Assume we have a program $S$ for the room ventilation controller.
- Assume we can observe at well-defined points in time the conditions $b$, off, on, go, stop when the software runs.
- Then the behaviour $[S]$ of $S$ can be viewed as computation paths of the form

$$\sigma_0 \xrightarrow{\alpha} \sigma_1 \xrightarrow{\alpha} \sigma_2 \cdots$$

where each $\sigma_i$ is a valuation of $b$, off, on, go, stop, i.e. $\sigma_i : \{b, \text{off}, \text{on}, \text{go}, \text{stop}\} \rightarrow \mathbb{B}$. 

---

---
Software Satisfies Software Specification: Another Example,

**Software Specification**

\[ \mathcal{S} : \]

- Assume we can observe at well-defined points in time the observables relevant for the LSC (conditions and messages) when the software \( S \) runs.
- Then the behaviour \( \llbracket S \rrbracket \) of \( S \) can be viewed as computation paths over the LSC's observables.
- And then we can relate \( S \) to \( \mathcal{S} \).

Define: \( (S, [\cdot]) \in \llbracket \mathcal{S} \rrbracket \) if and only if

- \texttt{tja...} (in a minute)

**Software**

Back to LSCs vs. Software
A software $S$ is called compatible with LSC $L$ over $C$ and $E$ if and only if

- $\Sigma = (C \rightarrow B), C \subseteq C$, i.e. the states comprise valuations of the conditions in $C$,
- $A = (B \rightarrow B), E_{I}^{\Sigma} \subseteq B$, i.e. the events comprise valuations of $E_{I}^{\Sigma}$.

A computation path $\pi = \sigma_{0}^{\alpha_{1}} \alpha_{2}^{\sigma_{2}} \cdots \in [S]$ of software $S$ induces the word

$$w(\pi) = (\sigma_{0} \cup \alpha_{1}), (\sigma_{1} \cup \alpha_{2}), (\sigma_{2} \cup \alpha_{3}), \ldots,$$

we use $W_{S}$ to denote the set of words induced by $[S]$, i.e.

$$W_{S} = \{w(\pi) \mid \pi \in [S]\}.$$
The Plan: A Formal Semantics for a Visual Formalism

Content

- Live Sequence Charts
  - TBA Construction
  - LSCs vs. Software
    - Software and Software Specification, formally
    - Software satisfies Software Specification
  - Full LSC (without pre-chart)
    - Activation Condition & Activation Mode
- (Slightly) Advanced LSC Topics
  - Full LSC with pre-chart
- LSCs in Requirements Engineering
  - strengthening existential LSCs (scenarios)
    into universal LSCs (requirements)
- LSCs in Quality Assurance
- Requirements Engineering Wrap-Up
  - Requirements Analysis in a Nutshell
  - Recall: Validation by Translation
- Outlook: Formal Methods in Design & QA
Activation Condition and Mode

Full LSC Syntax (without pre-chart)

A full LSC $\mathcal{L} = (MC, ac_0, am, \Theta_{\mathcal{L}})$ consists of

- (non-empty) main-chart $MC = ((L_M, \preceq_M, \sim_M), I_M, Msg_M, Cond_M, LocInv_M, \Theta_M)$,
- activation condition $ac_0 \in \Phi(\mathcal{C})$,
- strictness flag $strict$ (if false, $\mathcal{L}$ is permissive)
- activation mode $am \in \{ \text{initial, invariant} \}$,
- chart mode $\Theta_{\mathcal{L}} = \text{existential}$ ($\Theta_{\mathcal{L}} = \text{cold}$) or $\text{universal}$ ($\Theta_{\mathcal{L}} = \text{hot}$).
Software Satisfies LSC

Let \( S \) be a software which is compatible with LSC \( \mathcal{L} \) (without pre-chart).
We say software \( S \) satisfies LSC \( \mathcal{L} \), denoted by \( S \models \mathcal{L} \), if and only if

\[
\Theta_{\mathcal{L}} \quad \text{am = initial} \quad \text{am = invariant}
\]

\[
\begin{align*}
\text{cold} & : \quad \exists w \in W_G \bullet w^0 : \neg w_{\text{out}}(C_0) \\
& \land w^0 \models \psi_{\text{prog}}(0, C_0) \land w/1 \in \text{Lang}(\mathcal{L}(\mathcal{L})) \\
\text{hot} & : \quad \forall w \in W_G \bullet w^0 : \neg w_{\text{out}}(C_0) \\
& \Rightarrow w^0 \models \psi_{\text{prog}}(0, C_0) \land w/1 \in \text{Lang}(\mathcal{L}(\mathcal{L}))
\end{align*}
\]

where and \( C_0 \) is the minimal (or instance heads) cut of the main-chart.

Software Satisfies LSC

Let \( S \) be a software which is compatible with LSC \( \mathcal{L} \) (without pre-chart).
We say software \( S \) satisfies LSC \( \mathcal{L} \), denoted by \( S \models \mathcal{L} \), if and only if

\[
\Theta_{\mathcal{L}} \quad \text{am = initial} \quad \text{am = invariant}
\]

\[
\begin{align*}
\text{cold} & : \quad \exists w \in W_G \bullet w^0 : \neg w_{\text{out}}(C_0) \\
& \land w^0 \models \psi_{\text{prog}}(0, C_0) \land w/1 \in \text{Lang}(\mathcal{L}(\mathcal{L})) \\
\text{hot} & : \quad \forall w \in W_G \bullet w^0 : \neg w_{\text{out}}(C_0) \\
& \Rightarrow w^0 \models \psi_{\text{prog}}(0, C_0) \land w/1 \in \text{Lang}(\mathcal{L}(\mathcal{L}))
\end{align*}
\]

where and \( C_0 \) is the minimal (or instance heads) cut of the main-chart.

Software \( S \) satisfies a set of LSCs \( \mathcal{L}_1, \ldots, \mathcal{L}_n \) if and only if \( S \models \mathcal{L}_i \) for all \( 1 \leq i \leq n \).
Example: Vending Machine

- **Positive scenario:** Buy a Softdrink
  We (only) accept the software if it is possible to buy a softdrink.
  (i) Insert one 1 euro coin.
  (ii) Press the 'softdrink' button.
  (iii) Get a softdrink.

- **Positive scenario:** Get Change
  We (only) accept the software if it is possible to get change.
  (i) Insert one 50 cent and one 1 euro coin.
  (ii) Press the 'softdrink' button.
  (iii) Get a softdrink.
  (iv) Get 50 cent change.

- **Requirement:** Perform Self-Test on Power-on
  We (only) accept the software if it always performs a self-test on power-on.
  (i) Check water dispenser.
  (ii) Check softdrink dispenser.
  (iii) Check tea dispenser.
(Slightly) Advanced LSC Topics
A full LSC \( \mathcal{L} = (PC, MC, ac_0, am, \Theta_{\mathcal{L}}) \) consists of

- **pre-chart** \( PC = ((\mathcal{L}_P, \leq_P, \sim_P), I_P, \text{Msg}_P, \text{Cond}_P, \text{LocInv}_P, \Theta_P) \) (possibly empty),
- (non-empty) **main-chart** \( MC = ((\mathcal{L}_M, \leq_M, \sim_M), I_M, \text{Msg}_M, \text{Cond}_M, \text{LocInv}_M, \Theta_M) \),
- **activation condition** \( ac_0 \in \Phi(C) \),
- **strictness flag** `strict` (if false, \( \mathcal{L} \) is permissive)
- **activation mode** \( am \in \{\text{initial}, \text{invariant}\} \),
- **chart mode** existential \((\Theta_{\mathcal{L}} = \text{cold})\) or universal \((\Theta_{\mathcal{L}} = \text{hot})\).

**LSC Semantics with Pre-chart**

where \( C_0^P \) and \( C_0^M \) are the minimal (or instance heads) cuts of pre- and main-chart.
**Example: Vending Machine**

- **Requirement**: Buy Water
  We (only) accept the software if,
  (i) Whenever we insert 0.50 €,
  (ii) and press the 'water' button (and no other button),
  (iii) and there is water in stock,
  (iv) then we get water (and nothing else).

- **Negative scenario**: A Drink for Free
  We don't accept the software if it is possible to get a drink for free.
  (i) Insert one 1 euro coin.
  (ii) Press the 'softdrink' button.
  (iii) Do not insert any more money.
  (iv) Get two softdrinks.
LSCs in Requirements Analysis
One quite effective approach:

(i) **Approximate** the software requirements: ask for positive / negative **existential scenarios**.

- **Ask** the customer to describe example usages of the desired system.
  In the sense of: “If the system is not at all able to do this, then it’s not what I want.”
  (→ positive use-cases, existential LSC)

- **Ask** the customer to describe behaviour that **must not happen** in the desired system.
  In the sense of: “If the system does this, then it’s not what I want.”
  (→ negative use-cases, LSC with pre-chart and hot-false)

(ii) **Refine** result into **universal scenarios** (and validate them with customer).

- **Investigate** preconditions, side-conditions, exceptional cases and corner-cases.
  (→ extend use-cases, refine LSCs with conditions or local invariants)

- **Generalise** into universal requirements, e.g., **universal LSCs**.

- **Validate** with customer using new positive / negative scenarios.
**Strengthening Scenarios Into Requirements**

- **Ask customer for (pos./neg.) scenarios, note down as existential LSCs:**

- **Strengthen into requirements, note down as universal LSCs:**

- **Re-Discuss** with customer using example words of the LSCs’ language.
LSCs vs. Quality Assurance

How to Prove that a Software Satisfies an LSC?

* Software S satisfies existential LSC $\mathcal{Z}$ if there exists $\pi \in \llbracket S \rrbracket$ such that $\mathcal{Z}$ accepts $w(\pi)$. Prove $S \models \mathcal{Z}$ by demonstrating $\pi$.

* Note: Existential LSCs* may hint at test-cases for the acceptance test!
  (+: as well as (positive) scenarios in general, like use-cases)
How to Prove that a Software Satisfies an LSC?

- Software $S$ satisfies existential LSC $L$ if there exists $\pi \in \llbracket S \rrbracket$ such that $L$ accepts $w(\pi)$. Prove $S \models L$ by demonstrating $\pi$.

- Note: Existential LSCs* may hint at test-cases for the acceptance test! (= as well as (positive) scenarios in general, like use-cases)

- Universal LSCs (and negative/anti-scenarios!) in general need an exhaustive analysis! (Because they require that the software never ever exhibits the unwanted behaviour.) Prove $S \not\models L$ by demonstrating one $\pi$ such that $w(\pi)$ is not accepted by $L$.

Pushing Things Even Further

(Harel and Marelly, 2003)
Tell Them What You’ve Told Them...

- **Live Sequence Charts** (if well-formed)
  - have an abstract syntax: instance lines, messages, conditions, local invariants; mode: hot or cold.

- From an abstract syntax, mechanically construct its **TBA**.

- An **LSC** is **satisfied** by a software $S$ if and only if
  - **existential** (cold):
    - there is a word induced by a computation path of $S$
    - which is accepted by the LSC’s pre/main-chart TBA.
  - **universal** (hot):
    - all words induced by the computation paths of $S$
    - are accepted by the LSC’s pre/main-chart TBA.

- **Pre-charts** allow us to
  - specify **anti-scenarios** ("this must not happen"),
  - contrain activation.

- **Method**:
  - discuss (anti-)scenarios with customer,
  - generalise into universal LSCs and re-validate.

Requirements Engineering Wrap-Up
Risks Implied by Bad Requirements Specifications

- **design and implementation.**
  - without specification, programmers may just “ask around” when in doubt, possibly yielding different interpretations → **difficult integration**

- **negotiation** (with customer, marketing department, or ...)

- **documentation,** e.g., the user’s manual.
  - without specification, the user’s manual author can only describe what the system does, not what it should do ("every observation is a feature")

- **later re-implementations.**
  - the new software may need to adhere to requirements of the old software; if not properly specified, the new software needs to be a 1:1 re-implementation of the old → **additional effort**

- **preparation of tests,**
  - without a description of allowed outcomes, tests are randomly searching for generic errors (like crashes) → **systematic testing hardly possible**

- **acceptance by customer,** resolving later objections or regress claims.
  - without specification, it is unclear at delivery time whether behaviour is an error (developer needs to fix) or correct (customer needs to accept and pay) → **nasty disputes, additional effort**

- **re-use,**
  - without specification, re-use needs to be based on re-reading the code → **risk of unexpected changes**
• Customers **may not know** what they want.
  • That’s in general not their “fault”!
  • Care for **tacit** requirements.
  • Care for **non-functional** requirements / constraints.

• For **requirements elicitation**, consider starting with
  • **scenarios** ("positive use case") and **anti-scenarios** ("negative use case")
  and elaborate corner cases.
  Thus, **use cases** can be **very useful** — use case **diagrams** not so much.

• Maintain a **dictionary** and high-quality descriptions.

• Care for **objectiveness** / **testability** early on.
  Ask for each requirements: what is the **acceptance test**?

• **Use formal notations**
  • to **fully understand requirements** (precision),
  • for **requirements analysis** (completeness, etc.),
  • to communicate with your developers.

• If in doubt, **complement** (formal) **diagrams with text**
  (as safety precaution, e.g., in lawsuits).

---

**Formalisation Validation**

Two broad directions:

- **Option 1**: teach formalism (usually not economic).
- **Option 2**: serve as translator / mediator.

---

1. Domain experts tell system scenario S (maybe keep back, whether allowed / forbidden).
2. FM expert translates system scenario to valuation σ.
3. FM expert evaluates DT on σ.
4. FM expert translates outcome to "allowed / forbidden by DT".
5. Compare expected outcome and real outcome.

**Recommendation**: (Course’s Manifesto?)

- Use formal methods for the **most important/intricate requirements**
  (formalising all requirements is in most cases **not possible**),
- Use the **most appropriate formalism** for a given task,
- Use formalisms that you **know** (really) well.
(Strong) Literature Recommendation

(Rupp and die SOPHiSTen, 2014)

Big-Picture Outlook
Example: Software Specification

Alphabet:
- $M$ – dispense cash only,
- $C$ – return card only,
- $MC$ – dispense cash and return card.

- **Customer 1:** “don’t care”
  \[ \mathcal{S}_1 = \left( M.C \mid C.M \mid M \right)^\omega \]
- **Customer 2:** “you choose, but be consistent”
  \[ \mathcal{S}_2 = (M.C)^\omega \text{ or } (C.M)^\omega \]
- **Customer 3:** “consider human errors”
  \[ \mathcal{S}_3 = (C.M)^\omega \]

Formal Methods in the Software Development Process
References

