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**
Pre-Charts (Again)

- Pre-Charts
  - Semantics, once again
  - Requirements Engineering with scenarios
  - Strengthening scenarios into requirements
- Software, formally
  - Software specification
  - Requirements Engineering, formally
  - Software implements specification
- LSCs vs. Software
  - Software implements LSCs
  - Scenarios and tests
  - Play In/Play Out
- Requirements Engineering Wrap-Up
Example: Vending Machine

- **Positive scenario**: Buy a Softdrink
  (i) Insert one 1 euro coin.
  (ii) Press the 'softdrink' button.
  (iii) Get a softdrink.

- **Positive scenario**: 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.

- **Negative scenario**: 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.

Pre-Charts

A full LSC $\mathcal{L} = (PC, MC, ac, am, \Theta_C)$ actually consists of
- **pre-chart** $PC = ((L_P, \leq_P, \sim_P), L_P, Msg_P, Cond_P, LocInv_P, \Theta_P)$ (pos. empty),
- **main-chart** $MC = ((L_M, \leq_M, \sim_M), L_M, Msg_M, Cond_M, LocInv_M, \Theta_M)$,
- activation condition $ac \in \Phi(C)$, and mode $am \in \{initial, invariant\}$,
- strictness flag $\text{strict}$, chart mode existential ($\Theta_C = \text{cold}$) or universal ($\Theta_C = \text{hot}$).

Concrete syntax:
### Pre-Charts

A full **LSC** \( \mathcal{L} = (PC, MC, ac, am, \Theta_{\mathcal{L}}) \) actually consists of:

- **pre-chart** \( PC = (\{L_r, \neg L_r, \neg \nu\}, \tau_r, \text{Mag}_r, \text{Cond}_r, \text{LocInv}_r, \Theta_{\mathcal{L}}) \) (poss. empty).
- **main-chart** \( MC = (\{L_M, \neg L_M, \neg \nu\}, \tau_M, \text{Mag}_M, \text{Cond}_M, \text{LocInv}_M, \Theta_{\mathcal{L}}) \).
- activation condition \( ac \in \Phi(C) \) and mode \( am \in \{\text{initial}, \text{invariant}\} \).
- strictness flag \( strict \), chart mode \( existential (\Theta_{\mathcal{L}} = \text{cold}) \) or \( universal (\Theta_{\mathcal{L}} = \text{hot}) \).

A set of words \( W \subseteq (C \to E)^* \) is accepted by \( \mathcal{L} \), denoted by \( W \models \mathcal{L} \), if and only if:

<table>
<thead>
<tr>
<th>( am = \text{initial} )</th>
<th>( am = \text{invariant} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w \in W, m \in K \bullet )</td>
<td>( w \in W, m \in K \bullet )</td>
</tr>
<tr>
<td>( w/1 } \wedge \nu \in \text{Lang}(\text{B}(PC)) )</td>
<td>( w/1 } \wedge \nu \in \text{Lang}(\text{B}(PC)) )</td>
</tr>
<tr>
<td>( w/m + 1 } \wedge \psi_{\psi_{\nu}(0, C_0^M)} )</td>
<td>( w/m + 1 } \wedge \psi_{\psi_{\nu}(0, C_0^M)} )</td>
</tr>
<tr>
<td>( w/m + 2 \in \text{Lang}(\text{B}(MC)) )</td>
<td>( w/m + 2 \in \text{Lang}(\text{B}(MC)) )</td>
</tr>
</tbody>
</table>

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

### Universal LSC: Example

(i) **trivially satisfy ‘by code’**

\( w = (\text{Cd}1), (\text{Cd}1), (\text{Cd}1), (\text{Cd}1), (\text{Cd}1), \ldots \)

(ii) **\( w = (\text{Cd}1), (\text{Cd}1), (\text{Eq}1), (\text{Eq}1), (\text{Eq}1), \ldots \)**

(iii) **satisfy chart (non-trivially)**

\( w = (\text{Cd}1), (\text{Cd}1), (\text{Eq}1), (\text{Eq}1), (\text{Eq}1), (\text{Eq}1), (\text{Eq}1), \ldots \)**

(distinguish by legal exit in main-chart, or not)
Universal LSC: Example

Requirements Engineering with Scenarios
One quite effective approach:

(i) **Approximate** the software requirements: ask for positive / negative existential scenarios.
(ii) **Refine** result into **universal scenarios** (and validate them with customer).

That is:

- 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-folks)
- 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 LSC’s language.
A requirements specification should be

- correct — it correctly represents the wishes/needs of the customer,
- complete — all requirements (existing in somebody's head, or a document, or ...) should be present,
- relevant — things which are not relevant to the project should not be constrained,
- consistent, free of contradictions — each requirement is compatible with all other requirements; otherwise the requirements are not realisable,
- neutral, abstract — a requirements specification does not constrain the realization more than necessary,
- traceable, comprehensible — the sources of requirements are documented, requirements are uniquely identifiable,
- testable, objective — the final product can objectively be checked for satisfying a requirement.

Correctness and completeness are defined relative to something which is usually only in the customer's head.
→ is is difficult to be sure of correctness and completeness.

"Dear customer, please tell me what is in your head!" is in almost all cases not a solution.
It's not unusual that even the customer does not precisely know...!
For example, the customer may not be aware of contradictions due to technical limitations.

Definition. [LSC Consistency] A set of LSCs \( \{ L_1, \ldots, L_n \} \) is called consistent if and only if there exists a set of words \( W \) such that \( \bigwedge_{i=1}^{n} W = \text{Lang}(L_i) \).

**Content**

- Pre-Charts
  - Semantics, once again
  - Requirements Engineering with scenarios
  - Strengthening scenarios into requirements
- Software, formally
  - Software specification
  - Requirements Engineering, formally
  - Software implements specification
- LSCs vs. Software
  - Software implements LSCs
  - Scenarios and tests
  - Play In/Play Out
- Requirements Engineering Wrap-Up
Software and Software Specification, formally

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

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

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 $\llbracket \cdot \rrbracket : S \mapsto \llbracket S \rrbracket$ is called interpretation of $S$. 
Example: Software, formally

Software is a finite description $S$ of a (possibly infinite) set $[S]$ of (finite or infinite) computation paths of the form $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \sigma_n$: state/configuration; $\alpha_i$: action/event.

- Java Programs.
  ```java
  public int f( int x, int y ) {
    x = x + y;
    y = x / 2;
    return y;
  }
  ```

- HTML.
  ```html
  <html>
  <head>
  <title>SWT 2016</title>
  </head>
  <body/>
  </html>
  ```
Example: Software, formally

Software is a finite description $S$ of a (possibly infinite) set $\llbracket S \rrbracket$ of (finite or infinite) computation paths of the form $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \sigma_i$: $\text{state/configuration}$; $\alpha_i$: $\text{action/event}$.

- Java Programs.
- HTML.
- etc. etc.

Software Specification, formally

**Definition.** A **software specification** is a finite description $\mathcal{S}$ of a (possibly infinite) set $\llbracket \mathcal{S} \rrbracket$ of softwares, i.e.

$$\llbracket \mathcal{S} \rrbracket = \{(S_1, \cdot_1), \ldots\}.$$  

The (possibly partial) function $\llbracket \cdot \rrbracket : \mathcal{S} \mapsto \llbracket \mathcal{S} \rrbracket$ is called **interpretation** of $\mathcal{S}$. 

\[\]
Example: Software Specification

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

- **Customer 1:** “don’t care”
  \[
  S_1 = (M.C | C.M | MC)^\omega
  \]

- **Customer 2:** “you choose, but be consistent”
  \[
  S_2 = (M.C)^\omega \text{ or } (C.M)^\omega
  \]

- **Customer 3:** “consider human errors”
  \[
  S_3 = (C.M)^\omega
  \]

More Examples: Software Specification, formally

A software specification is a finite description $\mathcal{S}$ of a set $\mathcal{S}$ of softwares $\{(S_1, \cdot), \ldots \}$. 

- Decision Tables.

<table>
<thead>
<tr>
<th>Room ventilation</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>button pressed?</td>
<td>$-$</td>
<td>$+$</td>
<td>$-$</td>
</tr>
<tr>
<td>ventilation off?</td>
<td>$-$</td>
<td>$+$</td>
<td>$-$</td>
</tr>
<tr>
<td>ventilation on?</td>
<td>$-$</td>
<td>$+$</td>
<td>$-$</td>
</tr>
<tr>
<td>start ventilation</td>
<td>$-$</td>
<td>$+$</td>
<td>$-$</td>
</tr>
<tr>
<td>stop ventilation</td>
<td>$-$</td>
<td>$+$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

\[
\sigma_0 \xrightarrow{r_1} \sigma_1 \xrightarrow{r_2} \sigma_2 \ldots
\]

If $\sigma_i \sqsubseteq \mathcal{F}_{\text{prev}}(C_j)$
then $\sigma_{i+1} \sqsubseteq \mathcal{F}_{\text{eff}}(C_j)$
A software specification is a finite description $\mathcal{S}$ of a set $\mathcal{S}$ of softwares $\{(S_1, \mathcal{L}_1), \ldots \}$.

- Decision Tables.
- LSCs.
- Global Invariants.

$$x \geq 0$$
A software specification is a finite description \( \mathcal{S} \) of a set \( \mathcal{S} \) of softwares \( \{ (S_1, [\cdot]_1), \ldots \} \).

- Decision Tables.
- LSCs.
- Global Invariants.
- State Machines.

\[ \text{later} \]

```java
public int f( int x, int y ) {
    x = x + y;
    y = x / 2;
    return y;
}
```
More Examples: Software Specification, formally

A software specification is a finite description $\mathcal{S}$ of a set $\{S_i, i \in \mathbb{N}\}$.

- Decision Tables.
- LSCs.
- Global Invariants.
- State Machines.
- Java Programs.
- etc. etc.

The Requirements Engineering Problem Formally

- **Requirements engineering:**
  Describe/specify the set of the allowed softwares as $\mathcal{S}$.
  **Note:** what is not constrained is allowed, usually!

- **Software development:**
  Create one software $S$ whose computation paths $\llbracket S \rrbracket$ are all allowed. i.e. $\llbracket S \rrbracket \in \mathcal{S}$.
  **Note:** different programs in different programming languages may describe the same $\llbracket S \rrbracket$.
  **Often allowed:** any refinement of $\mathcal{S}$ (→ in a minute; e.g. allow intermediate transitions).
$\mathcal{S} = \{(S_0, \llbracket \cdot \rrbracket_0)\}$

$\mathcal{S}_1$ implements $\mathcal{S}$ via $I$ and $M$

$\mathcal{S}_2$ implements $\mathcal{S}$ via $I'$ and $M'$

$LSCs$ vs. $Software$
A software $S$ is called compatible with LSC $L$ over $C$ and $E$ if and only if
- $\Sigma = (C \to B)$, i.e. the states are valuations of the conditions in $C$.
- $A \subseteq E_1$, i.e. the events are of the form $E!$, $E?$ (viewed as a valuation of $E!$, $E?$).

A computation path $\pi = \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \cdots \in [S]$ 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 $[S]$.

We say software $S$ satisfies LSC $L$ (without pre-chart), denoted by $S \models L$, if and only if

$\Theta_S |
\begin{align*}
\text{old} & \quad \text{inv} = \text{initial} & \quad \text{inv} = \text{invariant} \\
\exists w \in W_S \bullet w^0 \models ac \land \neg \psi_{\text{exit}}(C_0) \\
& \quad \land w^i \models \psi_{\text{prog}}(\emptyset, C_0) \land w/1 \in \text{Lang}(L(L')) \\
\forall w \in W_S \bullet w^0 \models ac \land \neg \psi_{\text{exit}}(C_0) \\
& \quad \land w^i \models \psi_{\text{prog}}(\emptyset, C_0) \land w/1 \in \text{Lang}(L(L'))
\end{align*}$

Software $S$ satisfies a set of LSCs $L_1, \ldots, L_n$ if and only if $S \models L_i$ for all $1 \leq i \leq n$.

How to Prove that a Software Satisfies an LSC?

- Software $S$ satisfies existential LSC $L$ if there exists $\pi \in [S]$ 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)

---

* LSCs as Software Specification

---
How to Prove that a Software Satisfies an LSC?

• Software $S$ satisfies existential LSC $\mathcal{L}$ if there exists $\pi \in [S]$ such that $\mathcal{L}$ accepts $w(\pi)$. Prove $S \models \mathcal{L}$ 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.)

Pushing It Even Further

(Harel and Marely, 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**.
- **Pre-charts** allow us to
  - specify **anti-scenarios** (“this must not happen”),
  - contrain activation.
- 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.
- **Method**:
  - discuss (anti-)scenarios with customer,
  - generalise into universal LSCs and re-validate.

Requirements Engineering Wrap-Up
**Example: Software Specification**

**Alphabet:**
- $M$ – dispense cash only.
- $C$ – return card only.
- $M \cdot C$ – dispense cash and return card.

- **Customer 1:** “don’t care”

$\mathcal{R}_1 = (M.C | C.M | M_C)^\omega$

- **Customer 2:** “you choose, but be consistent”

$\mathcal{R}_2 = (M.C)^\omega$ or $(C.M)^\omega$

- **Customer 3:** “consider human errors”

$\mathcal{R}_3 = (C.M)^\omega$
Tell Them What You’ve Told Them...

- A **Requirements Specification** should be
  - correct, complete, relevant, consistent, neutral, traceable, objective.

- **Requirements Representations** should be
  - easily understandable, precise, easily maintainable, easily usable.

- **Languages / Notations** for Requirements Representations:
  - Natural Language Patterns
  - **Decision Tables**
  - User Stories
  - Use Cases
  - **Live Sequence Charts**

- **Formal representations**
  - can be very **precise**, objective, testable,
  - can be **analysed** for, e.g., completeness, consistency
  - can be **verified** against a formal design description.

  (Formal) inconsistency of, e.g., a decision table **hints at** inconsistencies in the requirements.
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).

---

**Literature Recommendation**

(Rupp and die SOPHISTen, 2014)
References

