Requirements Engineering Wrap-Up

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)

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.

A full LSC

L = (PC, MC, ac, am, Θ)

• pre-chart
  PC = ((LP, ⪯P, ∼P), IP, MsgP, CondP, LocInvP, ΘP), (poss. empty),
• main-chart
  MC = ((LM, ⪯M, ∼M), IM, MsgM, CondM, LocInvM, ΘM),
• activation condition
  ac ∈ Φ(C), and
• mode
  am ∈ {initial, invariant}
• strictness flag
  strict, chart mode
  existential (ΘL = cold) or
  universal (ΘL = hot).
Requirements on Requirements Specifications

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 realisation 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. → It 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.

\[ \text{LSC Consistency} \]

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

\[ \bigwedge_{i=1}^n W = \text{Lang}(L_i) \]

Software.

Definition.

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

\[ \sigma_0 \alpha_1 \rightarrow \sigma_1 \alpha_2 \rightarrow \sigma_2 \cdots \]

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 $\llbracket S \rrbracket$ of (finite or infinite) computation paths of the form

$\sigma_0 \alpha_1 \rightarrow \sigma_1 \alpha_2 \rightarrow \sigma_2 \cdot \cdot \cdot$.

$\sigma_i$: state/configuration;
$\alpha_i$: action/event.

Java Programs

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

Example: Software Specification

Alphabet:

• $M$ – dispense cash only,
• $C$ – return card only,
• $MC$ – dispense cash and return card.

Customer 1: "don't care"

$S_1 = (\text{M.C} | \text{| C.M } | \text{C.M})$...

Customer 2: "you choose, but be consistent"

$S_2 = (\text{M.C})$ or $\text{(C.M)}$

Customer 3: "consider human errors"

$S_3 = (\text{C.M})$

Decision Tables

$T$: room ventilation

<table>
<thead>
<tr>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>$\times$</td>
<td>$\times$</td>
</tr>
<tr>
<td>$\times$</td>
<td>$\times$</td>
<td>$\ast$</td>
</tr>
<tr>
<td>$\ast$</td>
<td>$\times$</td>
<td>$\ast$</td>
</tr>
<tr>
<td>go</td>
<td>start ventilation</td>
<td>stop</td>
</tr>
<tr>
<td>stop</td>
<td>stop ventilation</td>
<td>stop</td>
</tr>
</tbody>
</table>

More Examples: Software Specification, formally
A software specification is a finite description \( S \) of a set \( \{ (S_1, \llbracket \cdot \rrbracket_1), ..., \} \).

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

The Requirements Engineering Problem Formally

\[ (\Sigma \times A)^\omega \]
all computation paths over \( \Sigma \) and \( A \), aka. chaos requirements, all these computation paths are allowed (maybe including refinements) one software (\( = \) set of computation paths) which satisfies the requirements one software which does not satisfy the requirements.

- Requirements engineering: Describe/specify the set of the allowed softwares as \( S \).
  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. \( \llbracket S \rrbracket \in S \).
  Note: different programs in different programming languages may describe the same \( \llbracket S \rrbracket \).
  Often allowed: any refinement (\( \to \) in a minute; e.g. allow intermediate transitions).
How to Prove that a Software Satisfies an LSC?

Exhaustive analysis

We say software

Software Specification vs. Software

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

Top 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

Example: Software Specification

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

Customer 1: "don’t care"
S₁ = (M.C⏐⏐⏐C.M⏐⏐⏐MC)ω

Customer 2: "you choose, but be consistent"
S₂ = (M.C)ω or (C.M)ω

Customer 3: "consider human errors"
S₃ = (C.M)ω

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).