Softwaretechnik / Software-Engineering

Lecture 10: Req. Eng. Wrap-Up / Architecture & Design

2017-06-22

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

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Requirements Engineering: Content

VL 6	• **Introduction**
	• **Requirements Specification**
	– Desired Properties
	– Kinds of Requirements
	– Analysis Techniques
⋮	• **Documents**
	– Dictionary, Specification
	• **Specification Languages**
	– Natural Language
VL 7	– Decision Tables
	– Syntax, Semantics
⋮	– Completeness, Consistency, …
VL 8	– Scenarios
⋮	– User Stories, Use Cases
	– Live Sequence Charts
VL 9	– Syntax, Semantics
⋮	• Definition: **Software** & **SW Specification**
VL 10	• **Wrap-Up**
⋮	

Content

- LSCs: **Automaton Construction**
- Excursion: **Symbolic Büchi Automata**
- **LSCs vs. Software**
- **Methodology**
  - Requirements Engineering with scenarios
  - Strengthening scenarions into requirements
- **Requirements Engineering Wrap-Up**

Topic Area Architecture & Design

- **Vocabulary**
  - (software) system, component, module, interface
  - design, architecture
- **Software Modelling**
  - model
  - views & viewpoints, the 4+1 view
  - model-driven software engineering

TBA Construction Principle

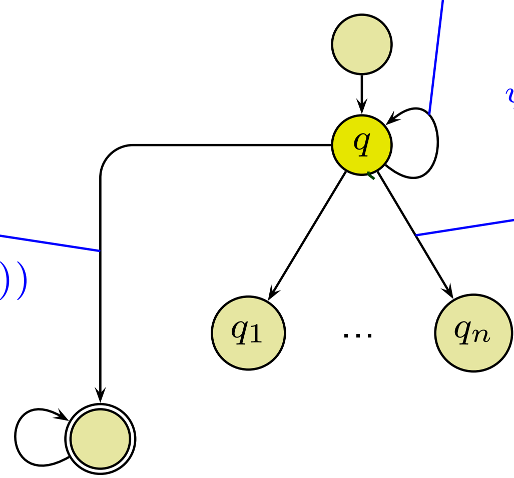
"Only" construct the transitions' labels:

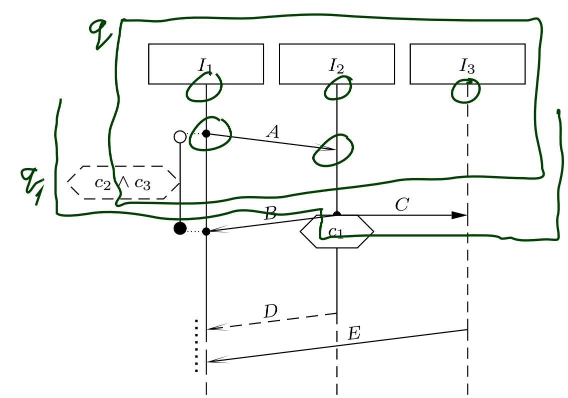
$$\rightarrow = \{(q, \psi_{loop}(q), q) \mid q \in Q\} \cup \{(q, \psi_{prog}(q, q'), q') \mid q \rightsquigarrow_{\mathcal{F}} q'\} \cup \{(q, \psi_{exit}(q), \mathcal{L}) \mid q \in Q\}$$

$$\psi_{loop}(q) = \overbrace{\psi^{\mathsf{Msg}}(q) \wedge \psi^{\mathsf{LocInv}}_{\mathsf{hot}}(q)}^{=:\psi^{\mathsf{hot}}_{loop}(q)} \wedge \psi^{\mathsf{LocInv}}_{\mathsf{cold}}(q)$$

$$\psi_{exit}(q) = (\psi^{\mathsf{hot}}_{loop}(q) \wedge \neg\psi^{\mathsf{LocInv}}_{\mathsf{cold}}(q)) \vee \bigvee_{1 \leq i \leq n} (\psi^{\mathsf{hot}}_{prog}(q, q_i) \wedge (\neg\psi^{\mathsf{LocInv},\bullet}_{\mathsf{cold}}(q, q_i) \vee \neg\psi^{\mathsf{Cond}}_{\mathsf{cold}}(q, q_i)))$$

$$\psi_{prog}(q, q_n) = \overbrace{\psi^{\mathsf{Msg}}(q, q_n) \wedge \psi^{\mathsf{Cond}}_{\mathsf{hot}}(q, q_n) \wedge \psi^{\mathsf{LocInv},\bullet}_{\mathsf{hot}}(q, q_n)}^{=:\psi^{\mathsf{hot}}_{prog}(q, q_n)} \wedge \psi^{\mathsf{Cond}}_{\mathsf{cold}}(q, q_n) \wedge \psi^{\mathsf{LocInv},\bullet}_{\mathsf{cold}}(q, q_n)$$





Loop Condition

$$\psi_{loop}(q) = \psi^{\mathsf{Msg}}(q) \wedge \psi^{\mathsf{LocInv}}_{\mathsf{hot}}(q) \wedge \psi^{\mathsf{LocInv}}_{\mathsf{cold}}(q)$$

- $\psi^{\mathsf{Msg}}(q) = \neg \bigvee_{1 \le i \le n} \psi^{\mathsf{Msg}}(q, q_i) \wedge \underbrace{\left(strict \implies \bigwedge_{\psi \in \mathcal{E}_{!?} \cap \mathsf{Msg}(\mathcal{L})} \neg\psi\right)}_{=:\psi_{\mathsf{strict}}(q)}$

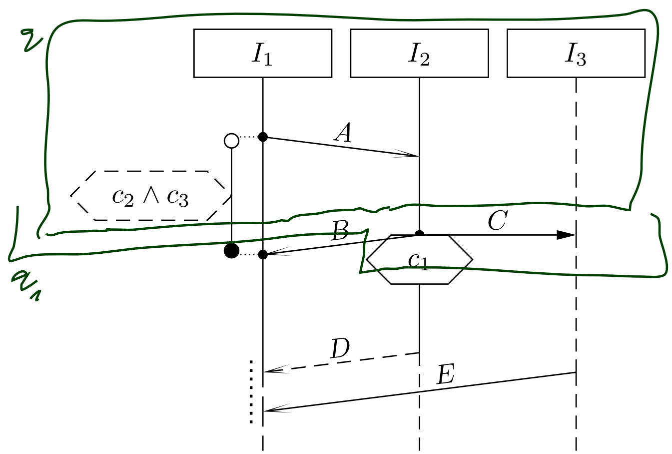
- $\psi^{\mathsf{LocInv}}_{\theta}(q) = \bigwedge_{\ell=(l,\iota,\phi,l',\iota') \in \mathsf{LocInv},\ \Theta(\ell)=\theta,\ \ell \text{ active at } q} \phi$

  A location l is called **front location** of cut C if and only if $\nexists\, l' \in \mathcal{L} \bullet l \prec l'$.

  Local invariant $(l_o, \iota_0, \phi, l_1, \iota_1)$ is **active** at cut (!) q
  if and only if $l_0 \preceq l \prec l_1$ for some front location l of cut q or $l = l_1 \wedge \iota_1 = \bullet$.

- $\mathsf{Msg}(\mathcal{F}) = \{E! \mid (l, E, l') \in \mathsf{Msg},\ l \in \mathcal{F}\} \cup \{E? \mid (l, E, l') \in \mathsf{Msg},\ l' \in \mathcal{F}\}$

- $\mathsf{Msg}(\mathcal{F}_1, \dots, \mathcal{F}_n) = \bigcup_{1 \le i \le n} \mathsf{Msg}(\mathcal{F}_i)$



Progress Condition

$$\psi_{prog}^{\mathsf{hot}}(q, q_i) = \psi^{\mathsf{Msg}}(q, q_{\cancel{n}\,i}) \wedge \psi_{\mathsf{hot}}^{\mathsf{Cond}}(q, q_{\cancel{n}\,i}) \wedge \psi_{\mathsf{hot}}^{\mathsf{LocInv},\bullet}(q_{\cancel{n}\,i})$$

- $\psi^{\mathsf{Msg}}(q, q_i) = \bigwedge_{\psi \in \mathsf{Msg}(q_i \setminus q)} \psi \wedge \bigwedge_{j \neq i} \bigwedge_{\psi \in (\mathsf{Msg}(q_j \setminus q)) \setminus \mathsf{Msg}(q_i \setminus q))} \neg\psi$
  $$\wedge \underbrace{\Big(strict \implies \bigwedge_{\psi \in (\mathcal{E}_{!?} \cap \mathsf{Msg}(\mathcal{L})) \setminus \mathsf{Msg}(\mathcal{F}_i)} \neg\psi\Big)}_{=:\psi_{\mathsf{strict}}(q, q_i)}$$

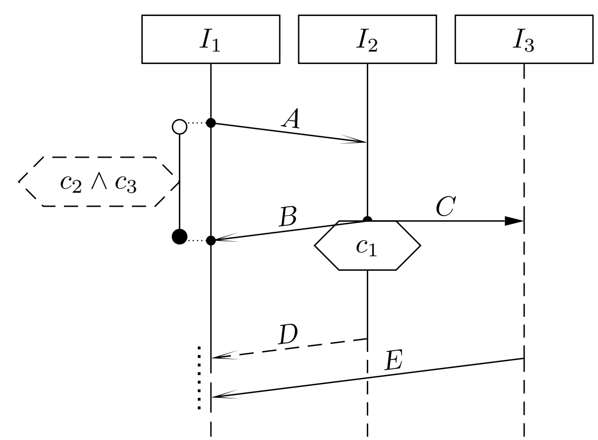
- $\psi_\theta^{\mathsf{Cond}}(q, q_i) = \bigwedge_{\gamma = (L, \phi) \in \mathsf{Cond},\ \Theta(\gamma) = \theta,\ L \cap (q_i \setminus q) \neq \emptyset} \phi$

- $\psi_\theta^{\mathsf{LocInv},\bullet}(q, q_i) = \bigwedge_{\lambda = (l, \iota, \phi, l', \iota') \in \mathsf{LocInv},\ \Theta(\lambda) = \theta,\ \lambda\ \bullet\text{-active at } q_i} \phi$

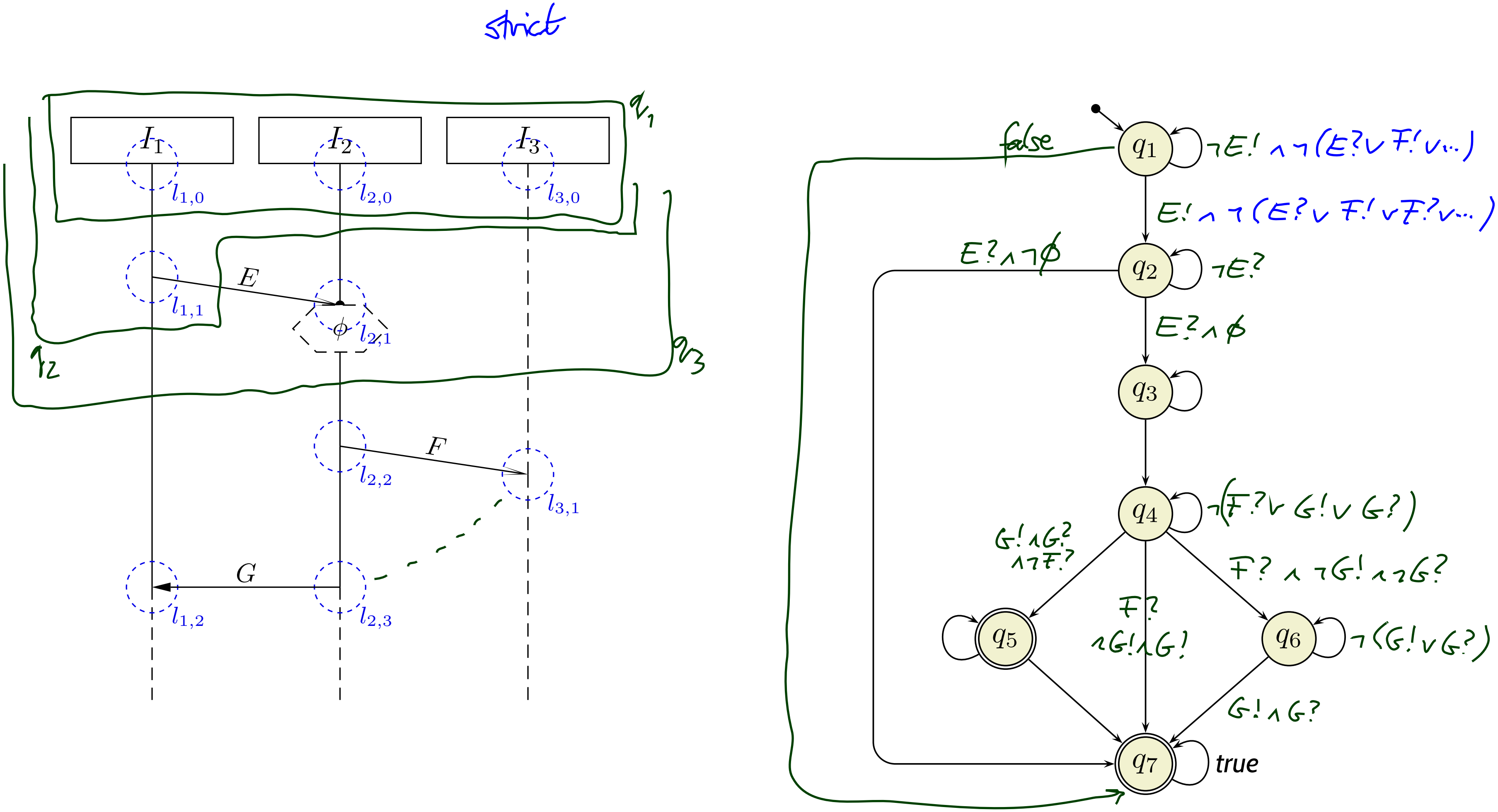
  Local invariant $(l_0, \iota_0, \phi, l_1, \iota_1)$ is **$\bullet$-active** at q if and only if

  - $l_0 \prec l \prec l_1$, or
  - $l = l_0 \wedge \iota_0 = \bullet$, or
  - $l = l_1 \wedge \iota_1 = \bullet$

  for some front location l of cut (!) q.



Example


strict
I_1
I_2
I_3
$l_{1,0}$
$l_{2,0}$
$l_{3,0}$
$l_{1,1}$
$l_{2,1}$
$l_{2,2}$
$l_{3,1}$
$l_{1,2}$
$l_{2,3}$
E
F
G
ϕ
q_1
q_2
q_3
q_1
q_2
q_3
q_4
q_5
q_6
q_7
false
$\neg E! \wedge \neg(E? \vee F! \vee ..)$
$E! \wedge \neg(E? \vee F! \vee F? \vee ..)$
$E? \wedge \neg\phi$
$\neg E?$
$E? \wedge \phi$
$\neg(F? \vee G! \vee G?)$
$G! \wedge G? \wedge \neg F?$
$F? \wedge \neg G! \wedge \neg G?$
$F? \wedge G! \wedge G!$
$\neg(G! \vee G?)$
$G! \wedge G?$
true

Content

- LSCs: **Automaton Construction**
- Excursion: **Symbolic Büchi Automata**
- **LSCs vs. Software**
- **Methodology**
  - Requirements Engineering with scenarios
  - Strengthening scenarions into requirements
- **Requirements Engineering Wrap-Up**

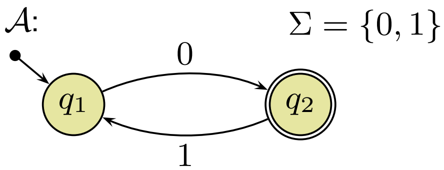
Topic Area Architecture & Design

- **Vocabulary**
  - (software) system, component, module, interface
  - design, architecture
- **Software Modelling**
  - model
  - views & viewpoints, the 4+1 view
  - model-driven software engineering

Excursion: Symbolic Büchi Automata

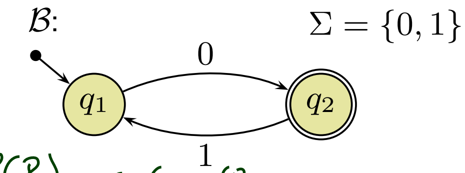
From Finite Automata to Symbolic Büchi Automata

$w_1 = 010101\ldots \in \mathcal{L}(\mathcal{B})$
$w_2 = 111\ldots \notin \mathcal{L}(\mathcal{B})$



Büchi

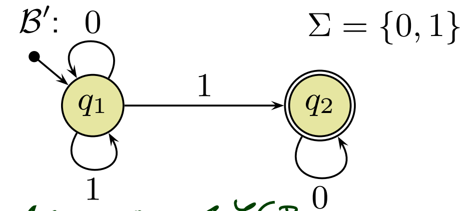
infinite words



$\mathcal{L}(\mathcal{A}) = 0.(1.0)^*$

$\mathcal{L}(\mathcal{B}) = 0.(10)^\omega$

$w_1 = 010 \in \mathcal{L}(\mathcal{A})$
$w_2 = 111 \notin \mathcal{L}(\mathcal{A})$

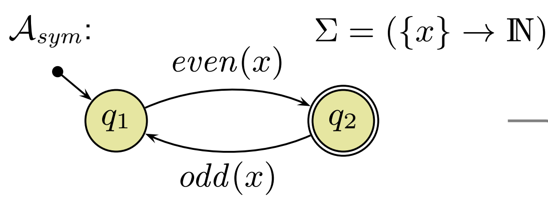


$w = 10000\ldots \in \mathcal{L}(\mathcal{B}')$

$\mathcal{L}(\mathcal{B}') = (0|1)^*.1.0^\omega$

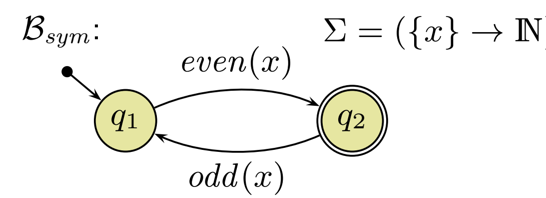
symbolic

symbolic



Büchi

infinite words



$w = (x\mapsto 0)\underbrace{(x\mapsto 1)}_{\in \{x\} \to \mathbb{N}}(x\mapsto 2)(x\mapsto 3) \in \mathcal{L}(\mathcal{A}_{sym})$

$(x\mapsto 1) \models odd(x)$

Symbolic Büchi Automata

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

$$\mathcal{B} = (\mathcal{C}_\mathcal{B}, Q, q_{ini}, \rightarrow, Q_F)$$

where

- $\mathcal{C}_\mathcal{B}$ is a set of atomic propositions,
- Q is a finite set of **states**,
- $q_{ini} \in Q$ is the initial state,
- $\rightarrow \subseteq Q \times \Phi(\mathcal{C}_\mathcal{B}) \times Q$ is the finite **transition relation**.
  Each transitions $(q, \psi, q') \in \rightarrow$ from state q to state q' is labelled with a formula $\psi \in \Phi(\mathcal{C}_\mathcal{B})$.
- $Q_F \subseteq Q$ is the set of **fair** (or accepting) states.

Run of TBA

Definition. Let $\mathcal{B} = (\mathcal{C}_\mathcal{B}, Q, q_{ini}, \to, Q_F)$ be a TBA and

$$w = \sigma_1, \sigma_2, \sigma_3, \dots \in (\Phi(\mathcal{C}_\mathcal{B}) \to \mathbb{B})^\omega$$

an infinite word, each letter is a valuation of $\Phi(\mathcal{C}_\mathcal{B})$.

An infinite sequence

$$\varrho = q_0, q_1, q_2, \ldots \in Q^\omega$$

of states is called **run** of $\mathcal{B}$ over w if and only if

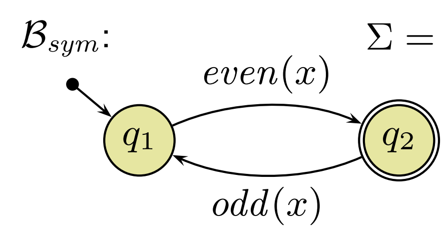
- $q_0 = q_{ini}$,
- for each $i \in \mathbb{N}_0$ there is a transition $(q_i, \psi_i, q_{i+1}) \in \to$ s.t. $\sigma_i \models \psi_i$.

$$w = \underbrace{(x \mapsto 0)}_{\models even(x)} \overbrace{(x \mapsto 1)}^{\models odd(x)} (x \mapsto 2) (x \mapsto 1) (x \mapsto 2) \cdots$$

$$\varrho = q_1, q_2, q_1, \cdots$$

Example:

$\Sigma = (\{x\} \to \mathbb{N})$



The Language of a TBA

Definition.
We say TBA $\mathcal{B} = (\mathcal{C}_\mathcal{B}, Q, q_{ini}, \to, Q_F)$ **accepts** the word

$$w = (\sigma_i)_{i \in \mathbb{N}_0} \in (\Phi(\mathcal{C}_\mathcal{B}) \to \mathbb{B})^\omega$$

if and only if $\mathcal{B}$ **has** a run

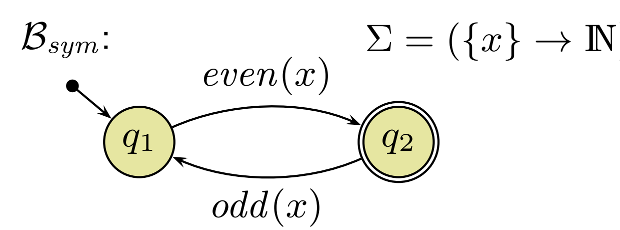
$$\varrho = (q_i)_{i \in \mathbb{N}_0}$$

over w such that
fair (or accepting) states are **visited infinitely often** by ϱ, i.e., such that

$$\forall\, i \in \mathbb{N}_0\, \exists\, j > i : q_j \in Q_F.$$

We call the set $Lang(\mathcal{B}) \subseteq (\Phi(\mathcal{C}_\mathcal{B}) \to \mathbb{B})^\omega$ of words that are accepted by $\mathcal{B}$ the **language of** $\mathcal{B}$.

Example:



LSCs vs. Software

LSCs as Software Specification

A software S is called **compatible** with LSC $\mathscr{L}$ over $\mathcal{C}$ and $\mathcal{E}$ is if and only if

- $\Sigma = (\mathcal{C} \to \mathbb{B})$, i.e. the **states** are valuations of the conditions in $\mathcal{C}$,
- $A \subseteq \mathcal{E}_{!?}$, i.e. the **events** are of the form $E!$, $E?$ (viewed as a valuation of $E!$, $E?$).

A computation path $\pi = \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \in [\![S]\!]$ of software S **induces** the word

$$w(\pi) = (\sigma_0 \cup \alpha_1), (\sigma_1 \cup \alpha_2), (\sigma_2 \cup \alpha_3), \dots,$$

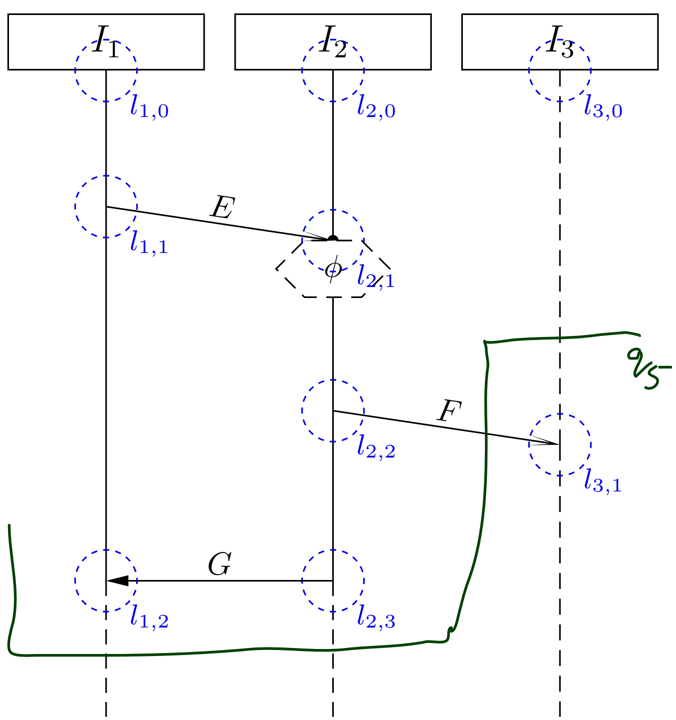
we use W_S to denote the set of words induced by $[\![S]\!]$.

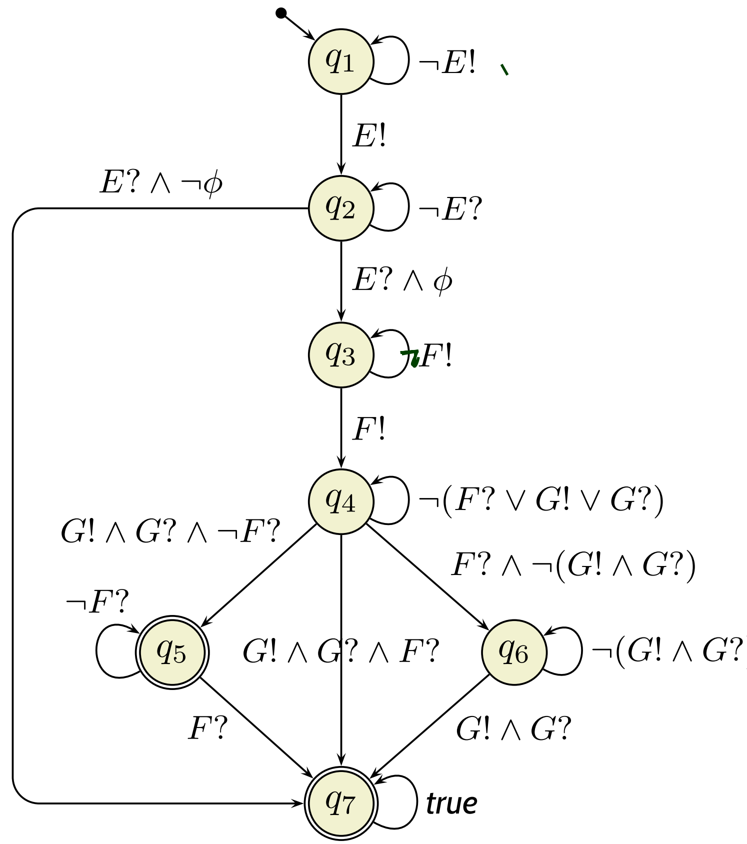
We say software S **satisfies** LSC $\mathscr{L}$ (without pre-chart), denoted by $S \models \mathscr{L}$, if and only if

$\Theta_{\mathscr{L}}$	am = initial	am = invariant
cold	$\exists\, w \in W_S \bullet w^0 \models ac \wedge \neg\psi_{exit}(C_0)$ $\wedge\, w^0 \models \psi_{prog}(\emptyset, C_0) \wedge w/1 \in Lang(\mathcal{B}(\mathscr{L}))$	$\exists\, w \in W_S\ \exists\, k \in \mathbb{N}_0 \bullet w^k \models ac \wedge \neg\psi_{exit}(C_0)$ $\wedge\, w^k \models \psi_{prog}(\emptyset, C_0) \wedge w/k+1 \in \underline{Lang(\mathcal{B}(\mathscr{L}))}$
hot	$\forall\, w \in W_S \bullet w^0 \models ac \wedge \neg\psi_{exit}(C_0)$ $\implies w^0 \models \psi_{prog}(\emptyset, C_0) \wedge w/1 \in Lang(\mathcal{B}(\mathscr{L}))$	$\forall\, w \in W_S\ \forall\, k \in \mathbb{N}_0 \bullet w^k \models ac \wedge \neg\psi_{exit}(C_0)$ $\implies w^k \models \psi^{\mathsf{Cond}}_{\mathsf{hot}}(\emptyset, C_0) \wedge w/k+1 \in Lang(\mathcal{B}(\mathscr{L}))$

Software S satisfies **a set of** LSCs $\mathscr{L}_1, \dots, \mathscr{L}_n$ if and only if $S \models \mathscr{L}_i$ for all $1 \leq i \leq n$.

Example: TBA vs. Computation Path





$$\pi_1 = \sigma_0 \xrightarrow{E!} \underbrace{\sigma_1}_{\models \phi} \xrightarrow{E?} \sigma_2 \xrightarrow{F!} \sigma_3 \xrightarrow{F? \wedge G! \wedge G?} \sigma_4 \longrightarrow \cdots \in Lang(\mathcal{L})$$

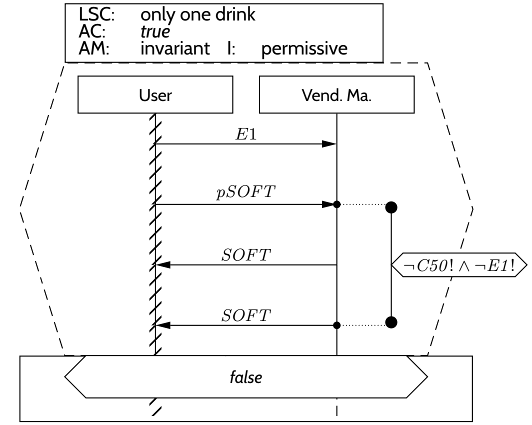
q_1 q_2 q_3 q_4 q_7 $q_7 \cdots$

$$\pi_4 = \sigma_0 \xrightarrow{E!} \underset{\phi}{\sigma_1} \xrightarrow{E?} \sigma_2 \xrightarrow{F!} \sigma_3 \overbrace{\xrightarrow{F? \wedge G!}}^{q_4} \sigma_4 \longrightarrow \cdots \notin Lang(\mathcal{L})$$

$$\pi_2 = \sigma_0 \xrightarrow{E!} \underbrace{\sigma_1}_{\not\models \phi} \xrightarrow{E?} \cdots \in Lang(\mathcal{L})$$

$$\pi_3 = \sigma_0 \xrightarrow{\tau} \sigma_1 \xrightarrow{\tau} \sigma_2 \xrightarrow{\tau} \cdots \notin Lang(\mathcal{L})$$

LSC Semantics (Corrected)



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

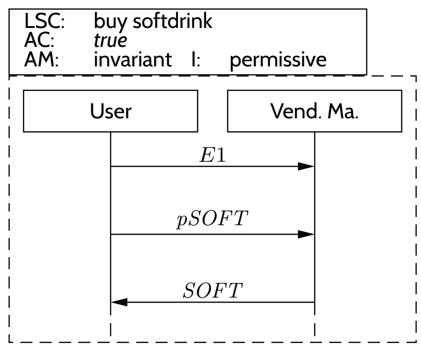
- **pre-chart** $PC = ((\mathcal{L}_P, \preceq_P, \sim_P), \mathcal{I}_P, \mathsf{Msg}_P, \mathsf{Cond}_P, \mathsf{LocInv}_P, \Theta_P)$ (poss. empty),
- **main-chart** $MC = ((\mathcal{L}_M, \preceq_M, \sim_M), \mathcal{I}_M, \mathsf{Msg}_M, \mathsf{Cond}_M, \mathsf{LocInv}_M, \Theta_M)$,
- **activation condition** $ac \in \Phi(\mathcal{C})$, and **mode** *am* $\in$ {initial, invariant},
- **strictness flag** $strict$, **chart mode existential** ($\Theta_\mathscr{L}$ = cold) or **universal** ($\Theta_\mathscr{L}$ = hot).

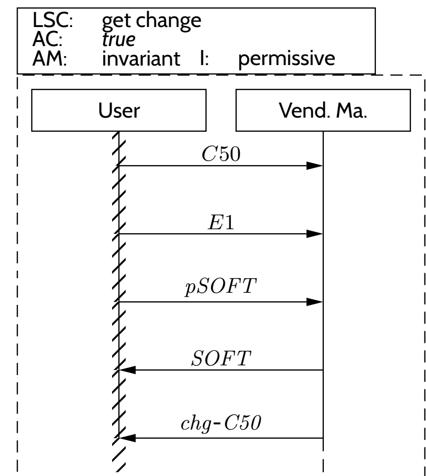
A **set of words** $W \subseteq (\mathcal{C} \to \mathbb{B})^\omega$ is **accepted** by $\mathscr{L}$, denoted by $W \models \mathscr{L}$, if and only if

	am = initial	am = invariant
$\Theta_\mathscr{L}$ = cold	$\exists\, w \in W\; \exists\, m \in \mathbb{N}_0 \bullet$ $\wedge\, w^0 \models ac \wedge \neg\psi_{exit}(C_0^P) \wedge \psi_{prog}(\emptyset, C_0^P)$ $\wedge\, w/1, \dots, w/m \in Lang(\mathcal{B}(PC))$ $\wedge\, w^{m+1} \models \neg\psi_{exit}(C_0^M)$ $\wedge\, w^{m+1} \models \psi_{prog}(\emptyset, C_0^M)$ $\wedge\, w/m+2 \in Lang(\mathcal{B}(MC))$	$\exists\, w \in W\; \exists\, k < m \in \mathbb{N}_0 \bullet$ $\wedge\, w^k \models ac \wedge \neg\psi_{exit}(C_0^P) \wedge \psi_{prog}(\emptyset, C_0^P)$ $\wedge\, w/k+1, \dots, w/m \in Lang(\mathcal{B}(PC))$ $\wedge\, w^{m+1} \models \neg\psi_{exit}(C_0^M)$ $\wedge\, w^{m+1} \models \psi_{prog}(\emptyset, C_0^M)$ $\wedge\, w/m+2 \in Lang(\mathcal{B}(MC))$
$\Theta_\mathscr{L}$ = hot	$\forall\, w \in W\; \forall\, m \in \mathbb{N}_0 \bullet$ $\wedge\, w^0 \models ac \wedge \neg\psi_{exit}(C_0^P) \wedge \psi_{prog}(\emptyset, C_0^P)$ $\wedge\, w/1, \dots, w/m \in Lang(\mathcal{B}(PC))$ $\wedge\, w^{m+1} \models \neg\psi_{exit}(C_0^M)$ $\implies w^{m+1} \models \psi_{prog}(\emptyset, C_0^M)$ $\wedge\, w/m+2 \in Lang(\mathcal{B}(MC))$	$\forall\, w \in W\; \forall\, k \leq m \in \mathbb{N}_0 \bullet$ $\wedge\, w^k \models ac \wedge \neg\psi_{exit}(C_0^P) \wedge \psi_{prog}(\emptyset, C_0^P)$ $\wedge\, w/k+1, \dots, w/m \in Lang(\mathcal{B}(PC))$ $\wedge\, w^{m+1} \models \neg\psi_{exit}(C_0^M)$ $\implies w^{m+1} \models \psi_{prog}(\emptyset, C_0^M)$ $\wedge\, w/m+2 \in Lang(\mathcal{B}(MC))$

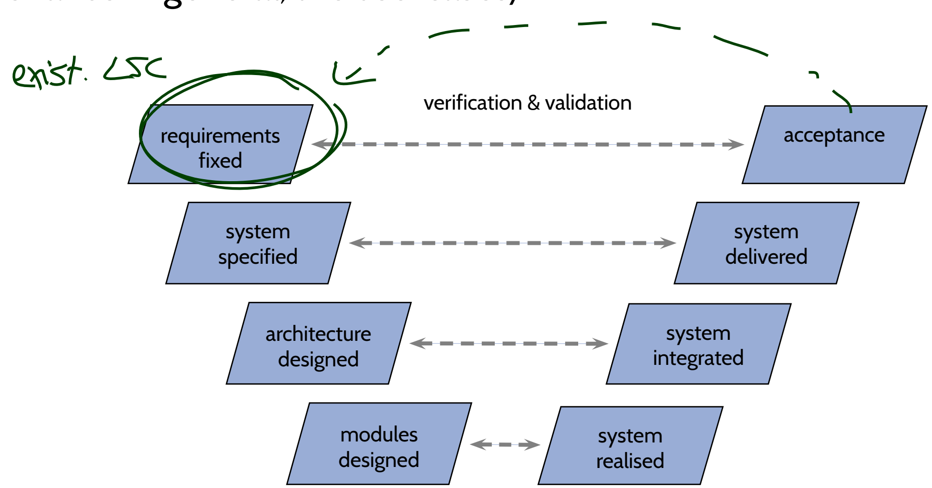
where C_0^P and C_0^M are the minimal (or **instance heads**) cuts of pre- and main-chart.

How to Prove that a Software Satisfies an LSC?

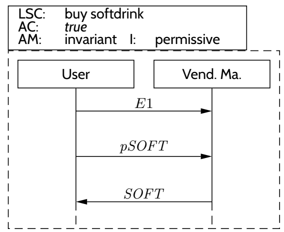


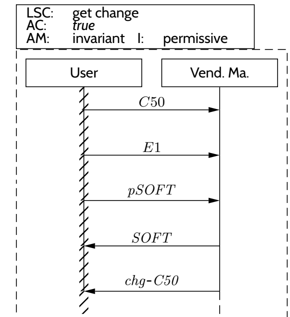


- Software S satisfies **existential** LSC $\mathscr{L}$ if there **exists** $\pi \in [\![S]\!]$ such that $\mathscr{L}$ accepts $w(\pi)$. Prove $S \models \mathscr{L}$ by demonstrating π.
- Note: **Existential** LSCs* may hint at **test-cases** for the **acceptance test**!
  ($*$: as well as (positive) scenarios in general, like use-cases)

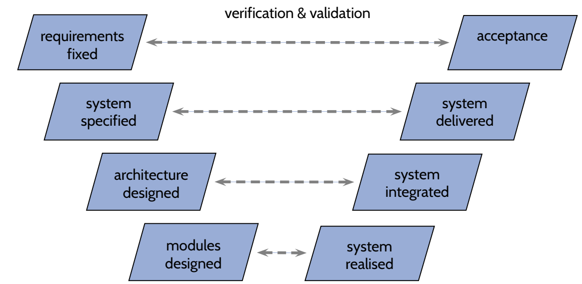


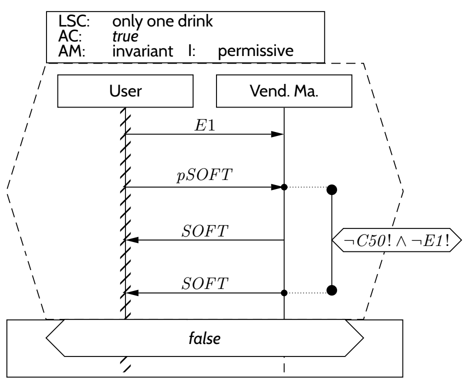
How to Prove that a Software Satisfies an LSC?

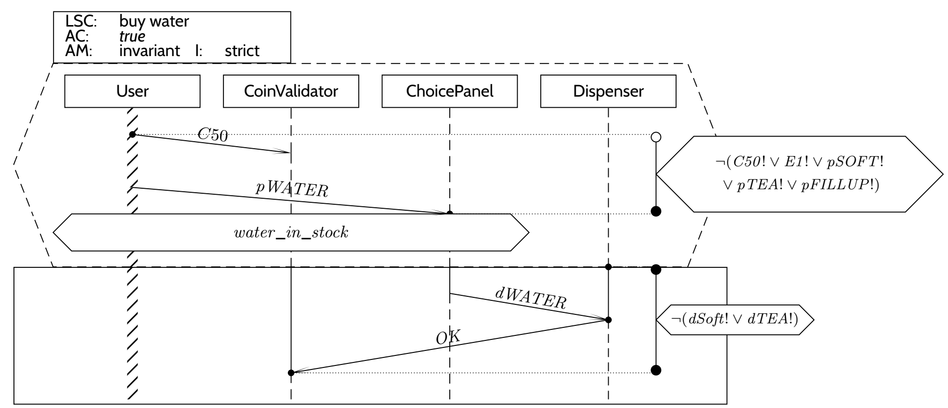




- Software S satisfies **existential** LSC $\mathscr{L}$ if there **exists** $\pi \in [\![S]\!]$ such that $\mathscr{L}$ accepts $w(\pi)$. Prove $S \models \mathscr{L}$ by demonstrating π.
- Note: **Existential** LSCs* may hint at **test-cases** for the **acceptance test**!
  ($*$: as well as (positive) scenarios in general, like use-cases)



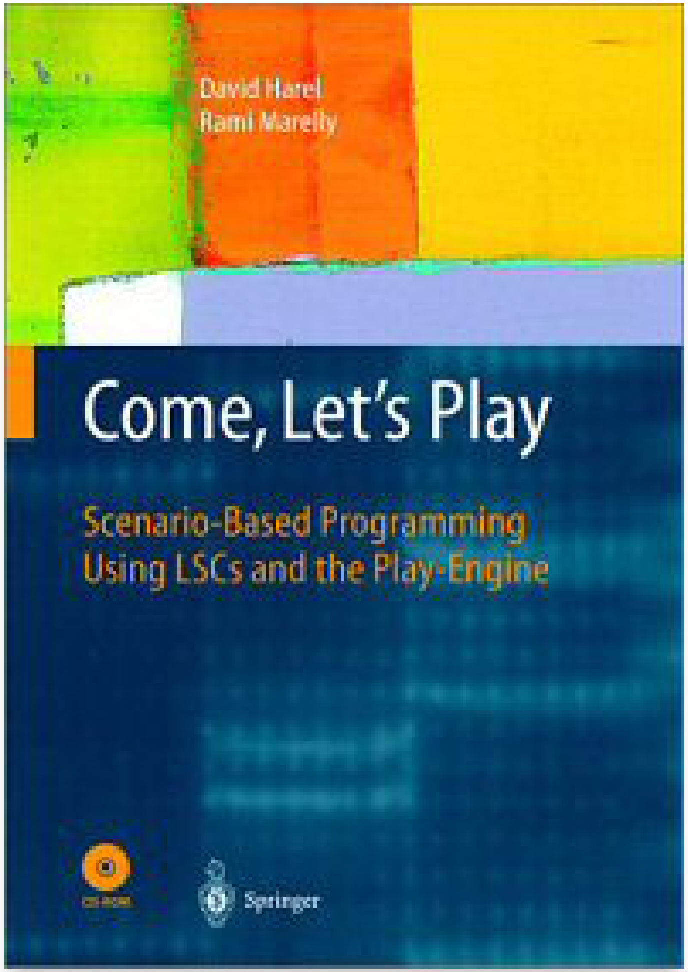




- **Universal** LSCs (and negative/anti-scenarios!) in general need an **exhaustive analysis**!
  (Because they require that the software never ever exhibits the unwanted behaviour.)

  Prove $S \not\models \mathscr{L}$ by demonstrating one π such that $w(\pi)$ **is not accepted** by $\mathscr{L}$.

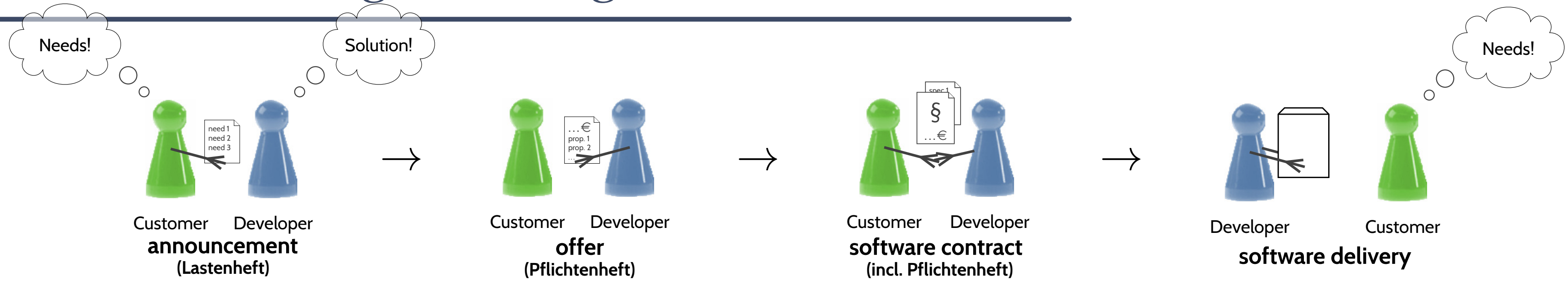
Pushing Things Even Further



(Harel and Marelly, 2003)

Requirements Engineering with Scenarios

Requirements Engineering with Scenarios



One quite effective approach:

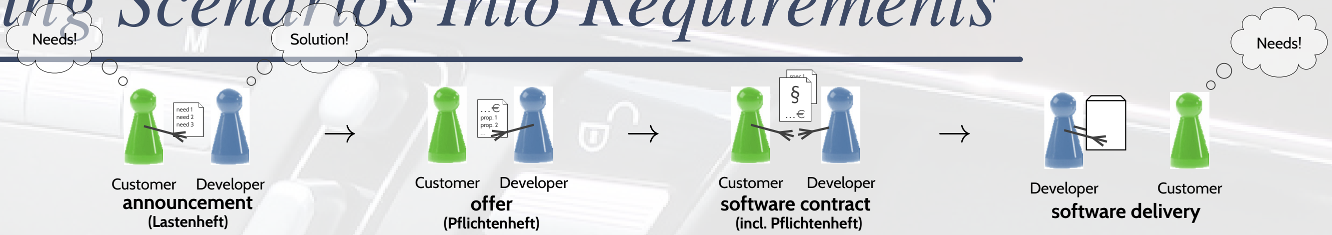
(i) **Approximate** the software requirements: ask for positive / negative **existential scenarios**.

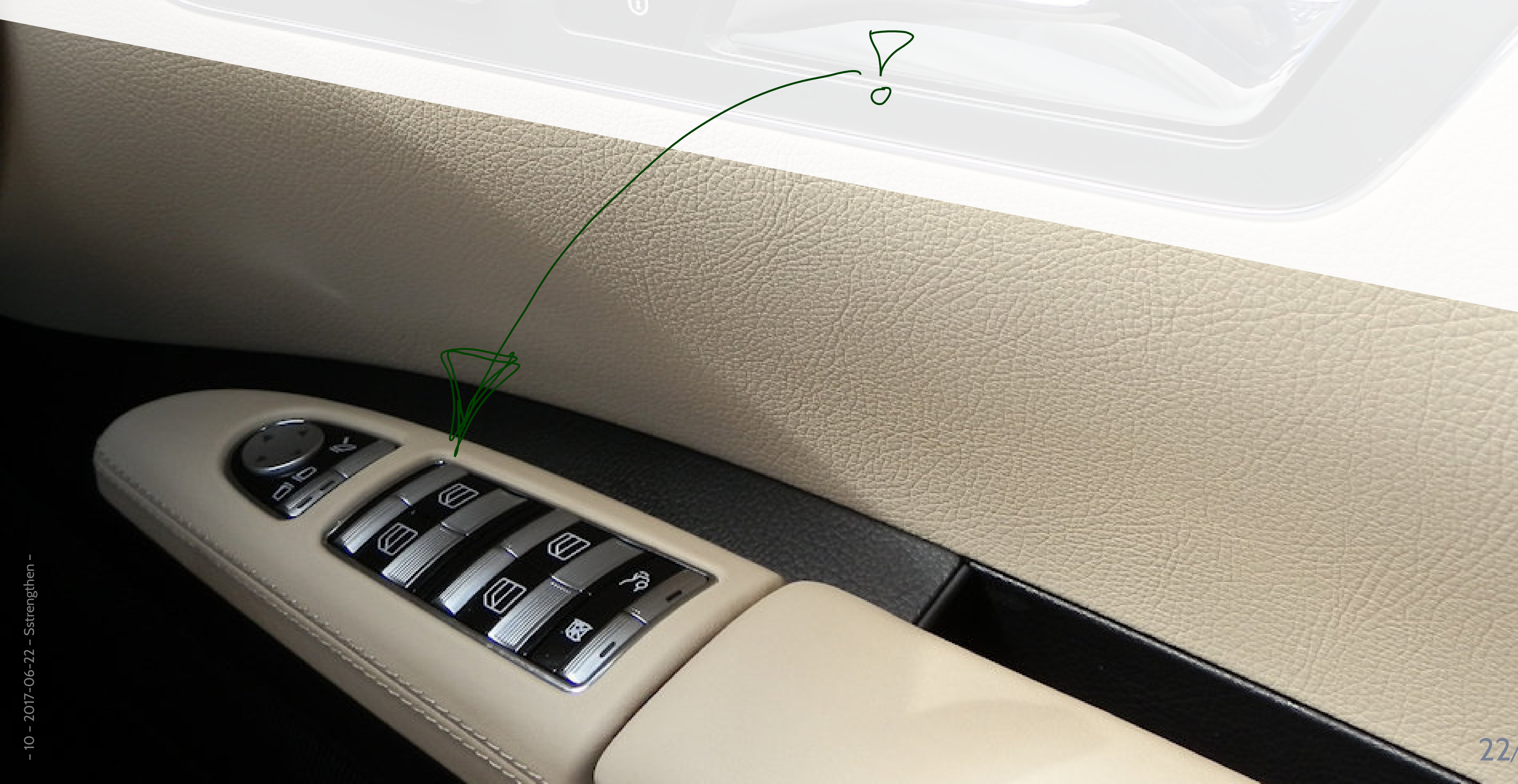
(ii) **Refine** result into **universal scenarios** (and validate them with customer).

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-*false*)
- **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.

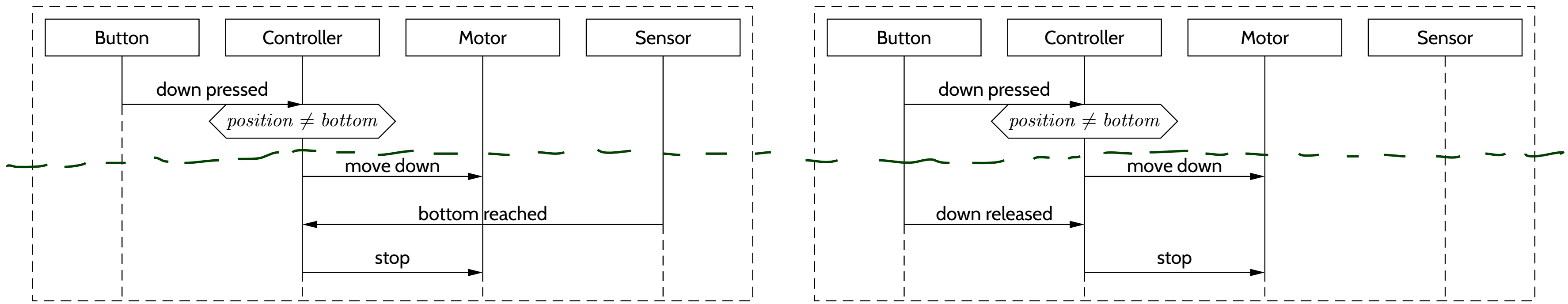
Strengthening Scenarios Into Requirements

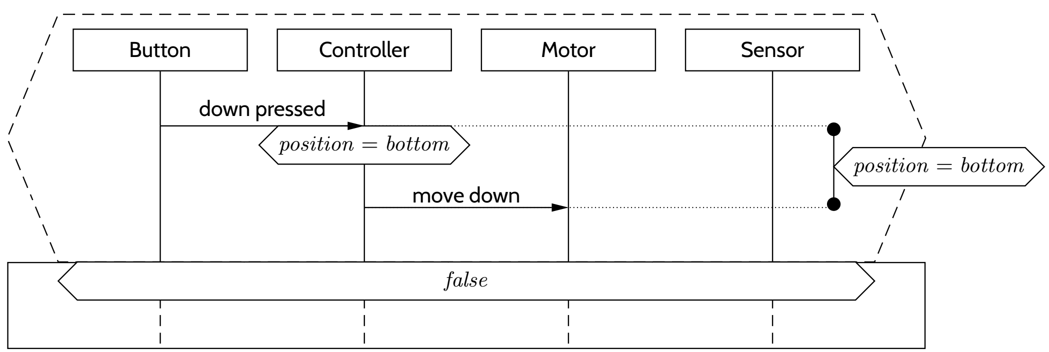




Strengthening Scenarios Into Requirements

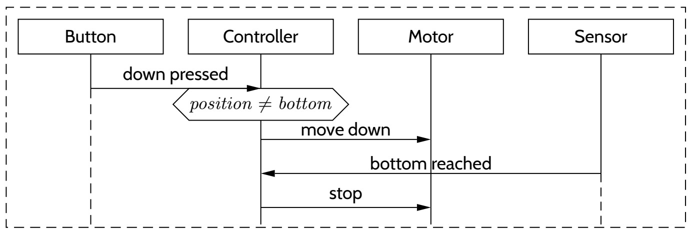
- **Ask customer for (pos./neg.) scenarios**, note down as existential LSCs:

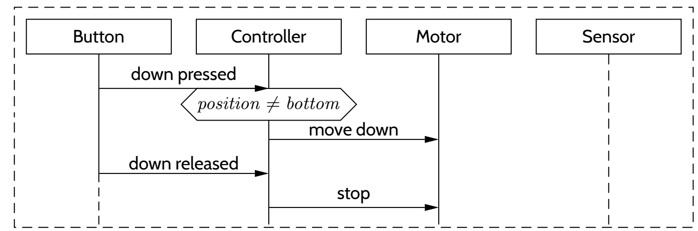


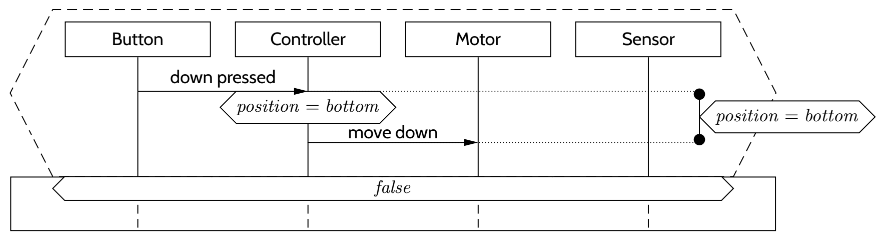


Strengthening Scenarios Into Requirements

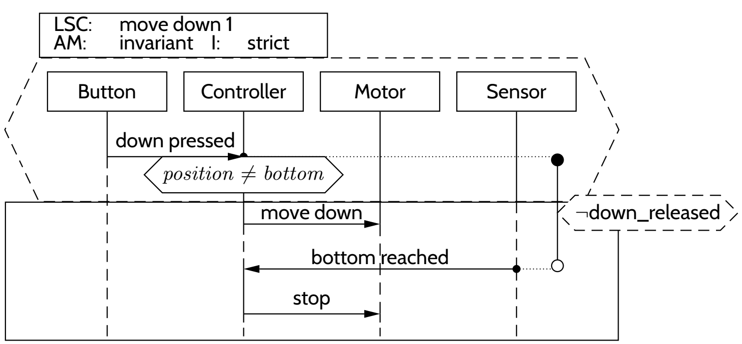
- **Ask customer for (pos./neg.) scenarios**, note down as existential LSCs:

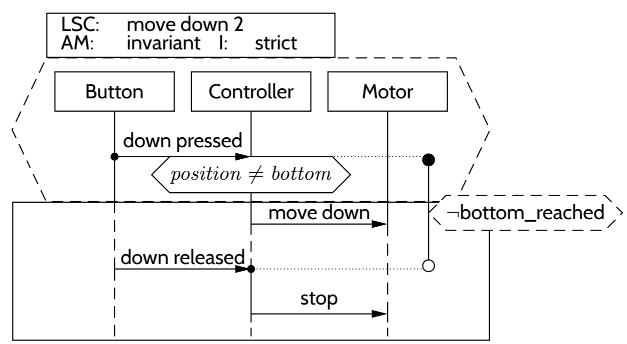


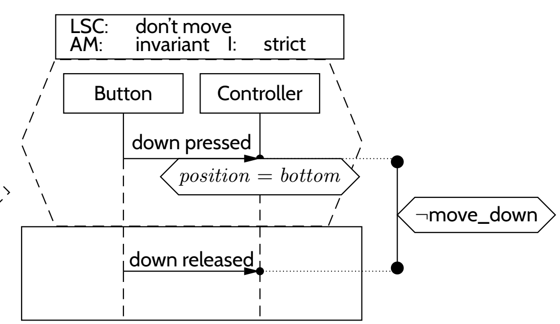




- **Strengthen into requirements**, note down as universal LSCs:







- **Re-Discuss** with customer using example words of the LSCs' language.

Analysing LSC Requiremnts

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**.
  $\rightarrow$ 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 $\{\mathscr{L}_1, \dots, \mathscr{L}_n\}$ is called **consistent** if and only if there exists a set of words W such that $\bigwedge_{i=1}^{n} W \models \mathscr{L}_i$.

Requirements Engineering Wrap-Up

Topic Area Requirements Engineering: Content

VL 6	• **Introduction**
	• **Requirements Specification**
	• Desired Properties
	• Kinds of Requirements
	• Analysis Techniques
⋮	• **Documents**
	• Dictionary, Specification
	• **Specification Languages**
	• Natural Language
VL 7	• Decision Tables
	• Syntax, Semantics
⋮	• Completeness, Consistency, …
VL 8	• Scenarios
⋮	• User Stories, Use Cases
	• Live Sequence Charts
VL 9	• Syntax, Semantics
⋮	• Definition: **Software** & **SW Specification**
VL 10 ⋮	• **Wrap-Up**

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.

Requirements Analysis in a Nutshell

- 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).

Example: Software Specification

Alphabet:

- M – dispense cash only,
- C – return card only,
- $\frac{M}{C}$ – dispense cash and return card.

- **Customer 1**: "don't care"

$$\mathscr{S}_1 = \left(M.C \middle| C.M \middle| \ \begin{matrix} M \\ C \end{matrix} \ \right)^\omega$$

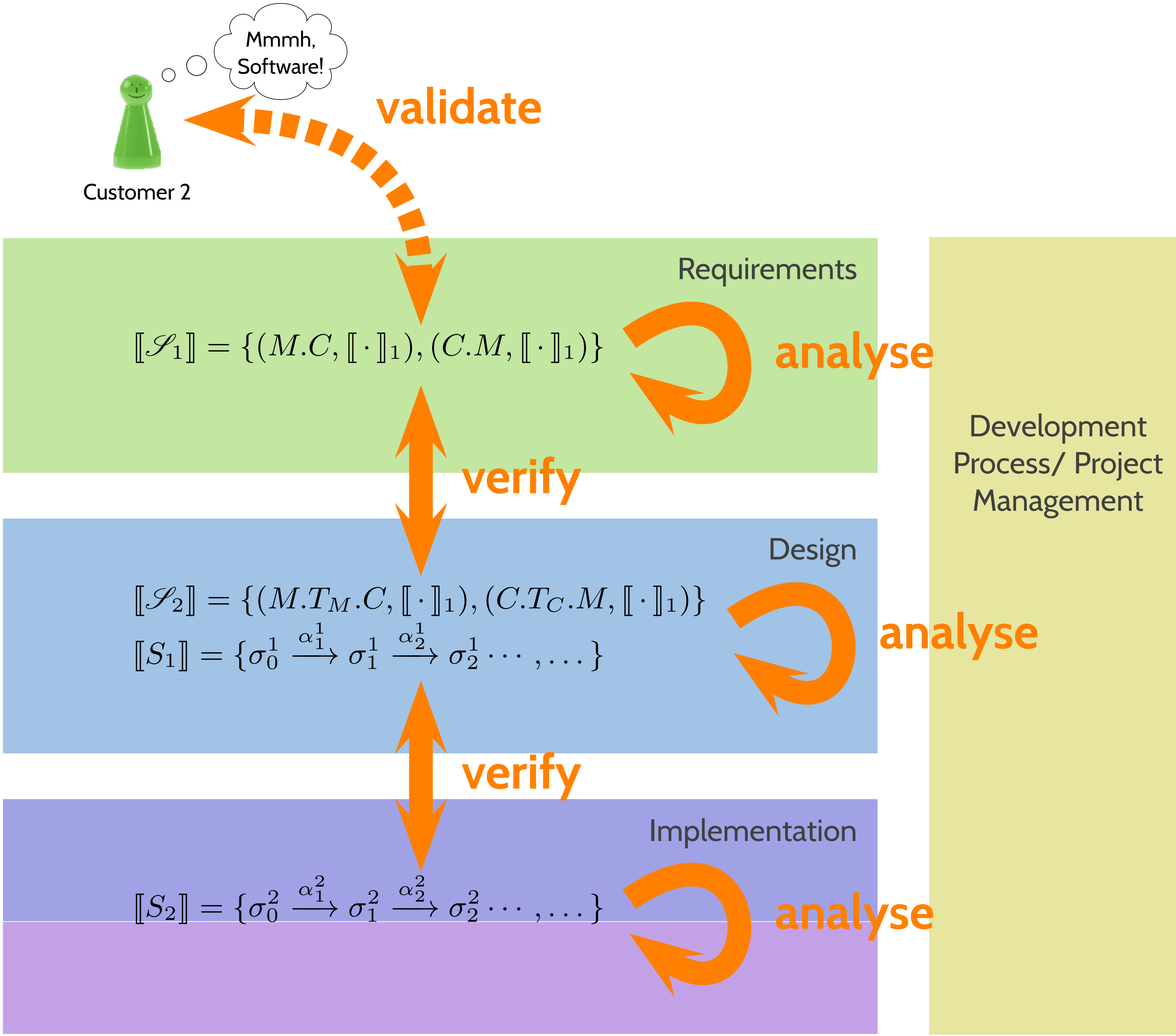
- **Customer 2**: "you choose, but be consistent"

$$\mathscr{S}_2 = (M.C)^\omega \text{ or } (C.M)^\omega$$

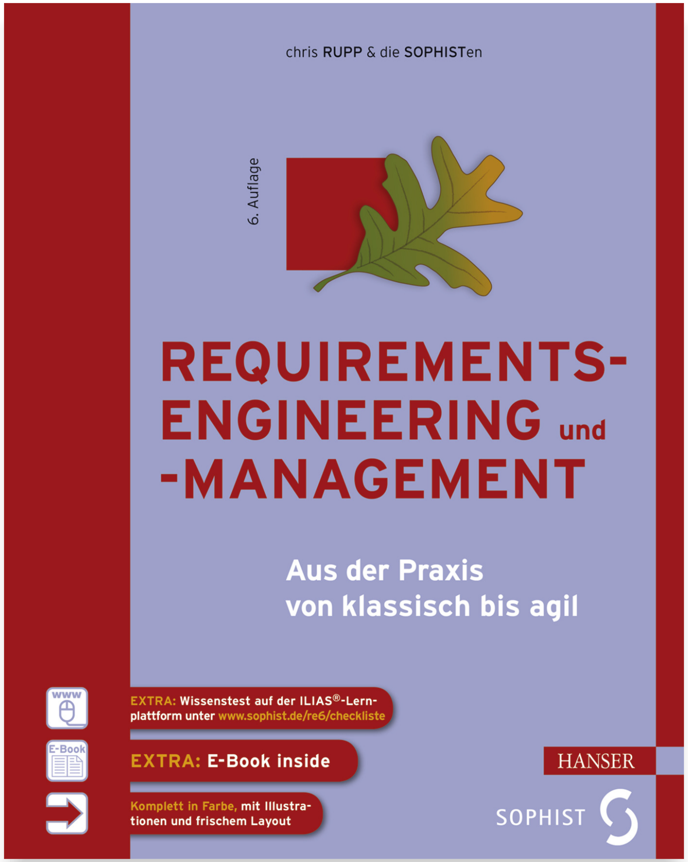
- **Customer 3**: "consider human errors"

$$\mathscr{S}_3 = (C.M)^\omega$$

Formal Methods in the Software Development Process



Literature Recommendation



(Rupp and die SOPHISTen, 2014)

Topic Area Architecture & Design: Content

VL 10

- **Introduction and Vocabulary**
- **Software Modelling**
  - (i) views and viewpoints, the 4+1 view
  - (ii) model-driven/-based software engineering
  - (iii) Unified Modelling Language (UML)

VL 11

  - (iv) **Modelling structure**
    - a) (simplified) class diagrams
    - b) (simplified) object diagrams
    - c) (simplified) object constraint logic (OCL)

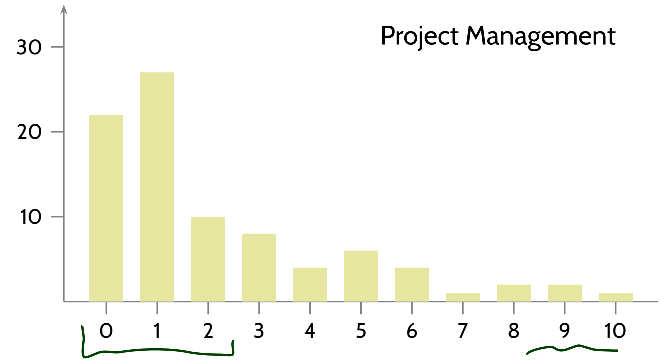
VL 12

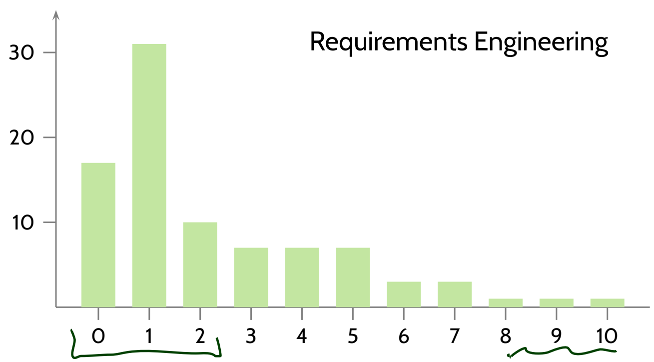
  - (v) **Principles of Design**
    - a) modularity
    - b) separation of concerns
    - c) information hiding and data encapsulation
    - d) abstract data types, object orientation
  - (vi) **Modelling behaviour**
    - a) communicating finite automata
    - b) Uppaal query language
    - c) basic state-machines
    - d) an outlook on hierarchical state-machines

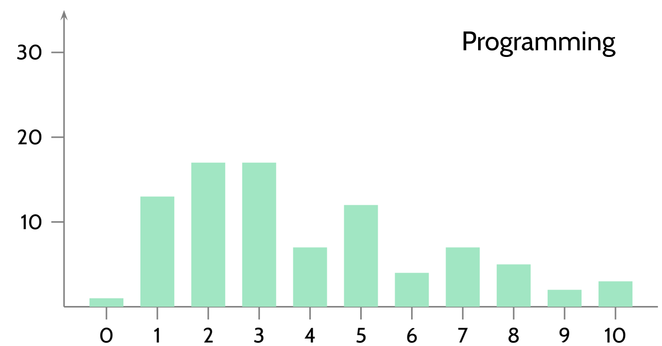
VL 13

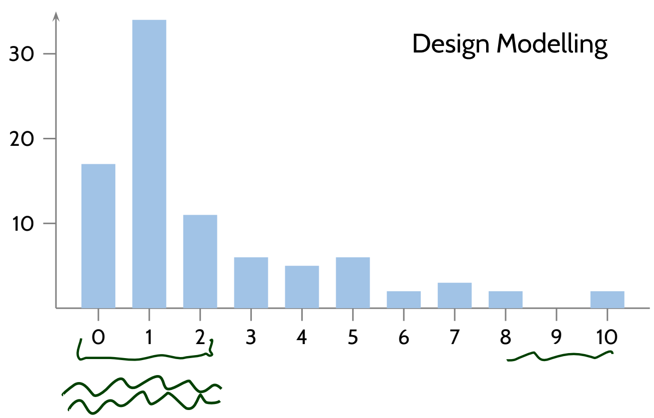
- **Design Patterns**

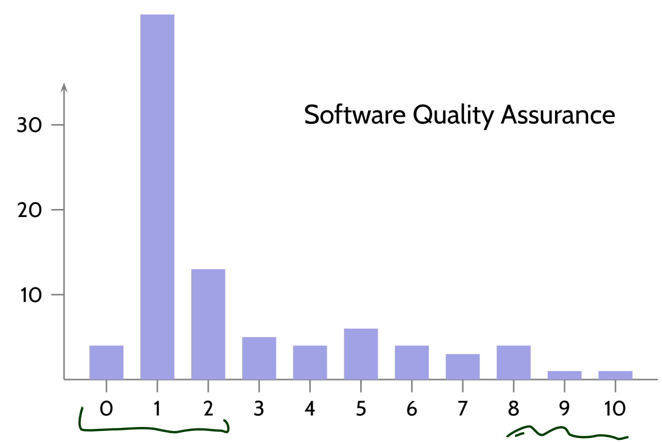
Survey: Previous Experience











Content

- LSCs: **Automaton Construction**
- Excursion: **Symbolic Büchi Automata**
- **LSCs vs. Software**
- **Methodology**
  - Requirements Engineering with scenarios
  - Strengthening scenarions into requirements
- **Requirements Engineering Wrap-Up**

Topic Area Architecture & Design

- **Vocabulary**
  - (software) system, component, module, interface
  - design, architecture
- **Software Modelling**
  - model
  - views & viewpoints, the 4+1 view
  - model-driven software engineering

Introduction

IEEE
Std 610.12-1990
(Revision and redesignation of
IEEE Std 792-1983)

IEEE Standard Glossary of Software Engineering Terminology

Sponsor

Standards Coordinating Committee of the Computer Society of the IEEE

Approved September 28, 1990

IEEE Standards Board

Abstract: IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology,* identifies terms currently in use in the field of Software Engineering. Standard definitions for those terms are established.
Keywords: Software engineering; glossary; terminology; definitions; dictionary

ISBN 1-55937-067-X

Vocabulary

system – A collection of components organized to accomplish a specific function or set of functions.
IEEE 1471 (2000)

software system – A set of software units and their relations, if they together serve a common purpose.
This purpose is in general complex, it usually includes, next to providing one (or more) executable program(s), also the organisation, usage, maintenance, and further development.
(Ludewig and Lichter, 2013)

component – One of the parts that make up a system. A component may be hardware or software and may be subdivided into other components.
IEEE 610.12 (1990)

software component – An architectural entity that
(1) encapsulates a subset of the system's functionality and/ or data,
(2) restricts access to that subset via an explicitly defined interface, and
(3) has explicitly defined dependencies on its required execution context.

(Taylor et al., 2010)

Vocabulary Cont'd

Vocabulary Cont'd

module– (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from an assembler, compiler, linkage editor, or executive routine..
(2) A logically separable part of a program. **IEEE 610.12 (1990)**

module– A set of operations and data visible from the outside only in so far as explicitly permitted by the programmers. (vs. interface) (Ludewig and Lichter, 2013)

interface– A boundary across which two independent entities meet and interact or communicate with each other. (Bachmann et al., 2002)

interface (of component)– The boundary between two communicating components. The interface of a component provides the services of the component to the component's environment and/or requires services needed by the component from the requirement. (Ludewig and Lichter, 2013)

Even More Vocabulary

Even More Vocabulary

design–
(1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component.
(2) The result of the process in (1). **IEEE 610.12 (1990)**

architecture– The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. **IEEE 1471 (2000)**

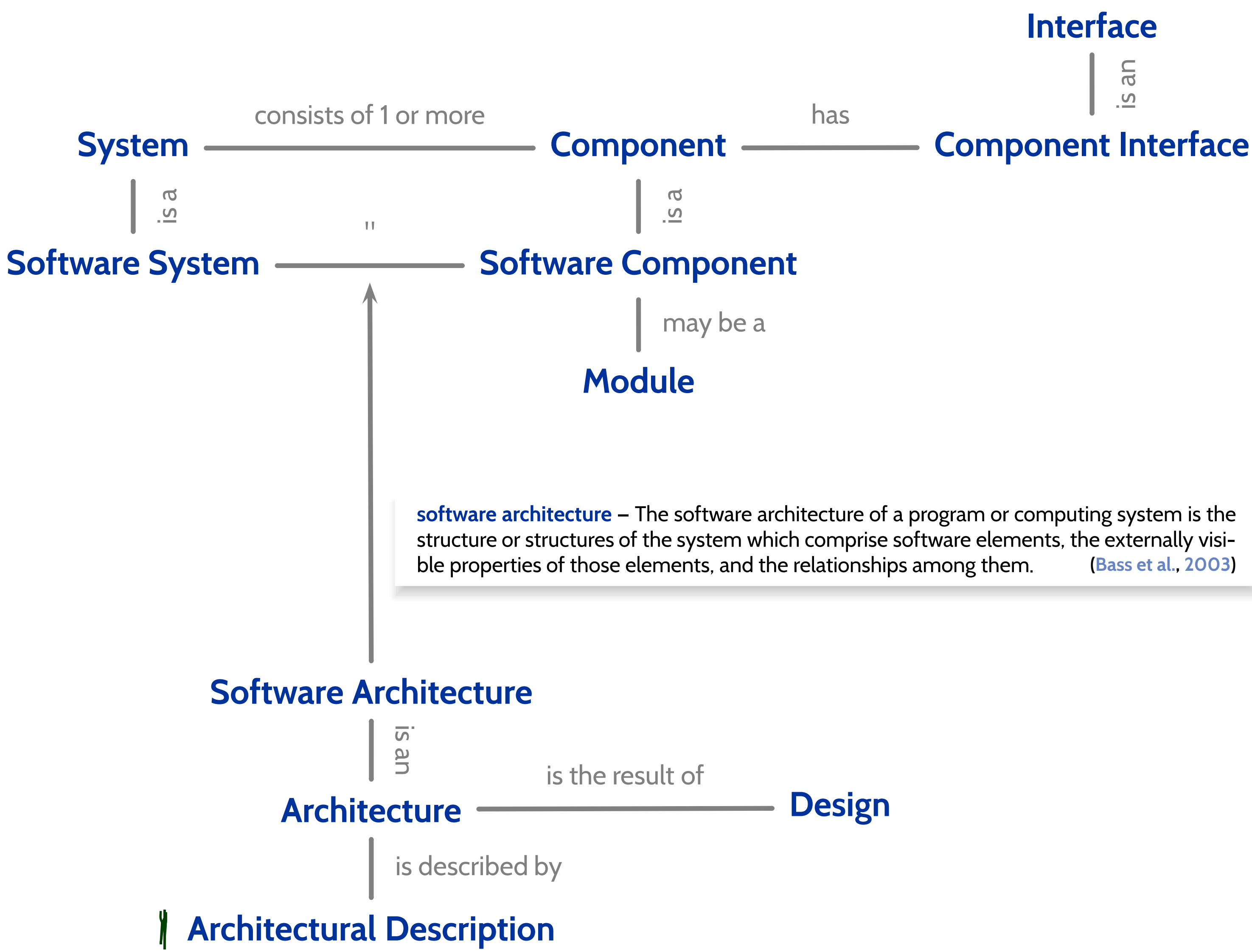
software architecture– The software architecture of a program or computing system is the structure or structures of the system which comprise software elements, the externally visible properties of those elements, and the relationships among them.

(Bass et al., 2003)

architectural description– A model – document, product or other artifact – to communicate and record a system's architecture. An architectural description conveys a set of views each of which depicts the system by describing domain concerns.

(Ellis et al., 1996)

Once Again, Please



software architecture – The software architecture of a program or computing system is the structure or structures of the system which comprise software elements, the externally visible properties of those elements, and the relationships among them. (Bass et al., 2003)

Goals and Relevance of Design

- The **structure** of something is the set of **relations between its parts**.
- Something not built from (recognisable) parts is called **unstructured**.

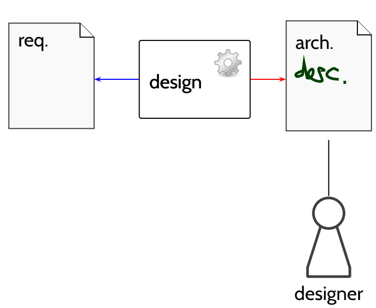
Design…

(i) **structures** a system into **manageable** units (yields software architecture),

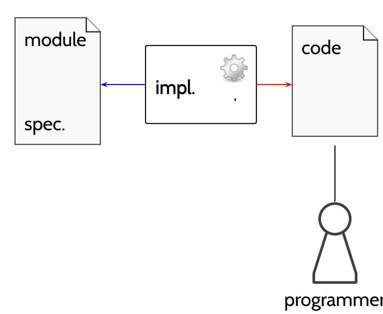
(ii) **determines** the approach for realising the required software,

(iii) provides **hierarchical structuring** into a **manageable** number of units at each hierarchy level.

Oversimplified process model “Design”:



design



implementation

Content

- LSCs: **Automaton Construction**
- Excursion: **Symbolic Büchi Automata**
- **LSCs vs. Software**
- **Methodology**
  - Requirements Engineering with scenarios
  - Strengthening scenarions into requirements
- **Requirements Engineering Wrap-Up**

Topic Area Architecture & Design

- **Vocabulary**
  - (software) system, component, module, interface
  - design, architecture
- **Software Modelling**
  - model
  - views & viewpoints, the 4+1 view
  - model-driven software engineering

Software Modelling

Model

Definition. (Folk) A **model** is an abstract, formal, mathematical representation or description of structure or behaviour of a (software) system.

Definition. (Glinz, 2008, 425)
A **model** is a concrete or mental image (Abbild) of something
or a concrete or mental archetype (Vorbild) for something.

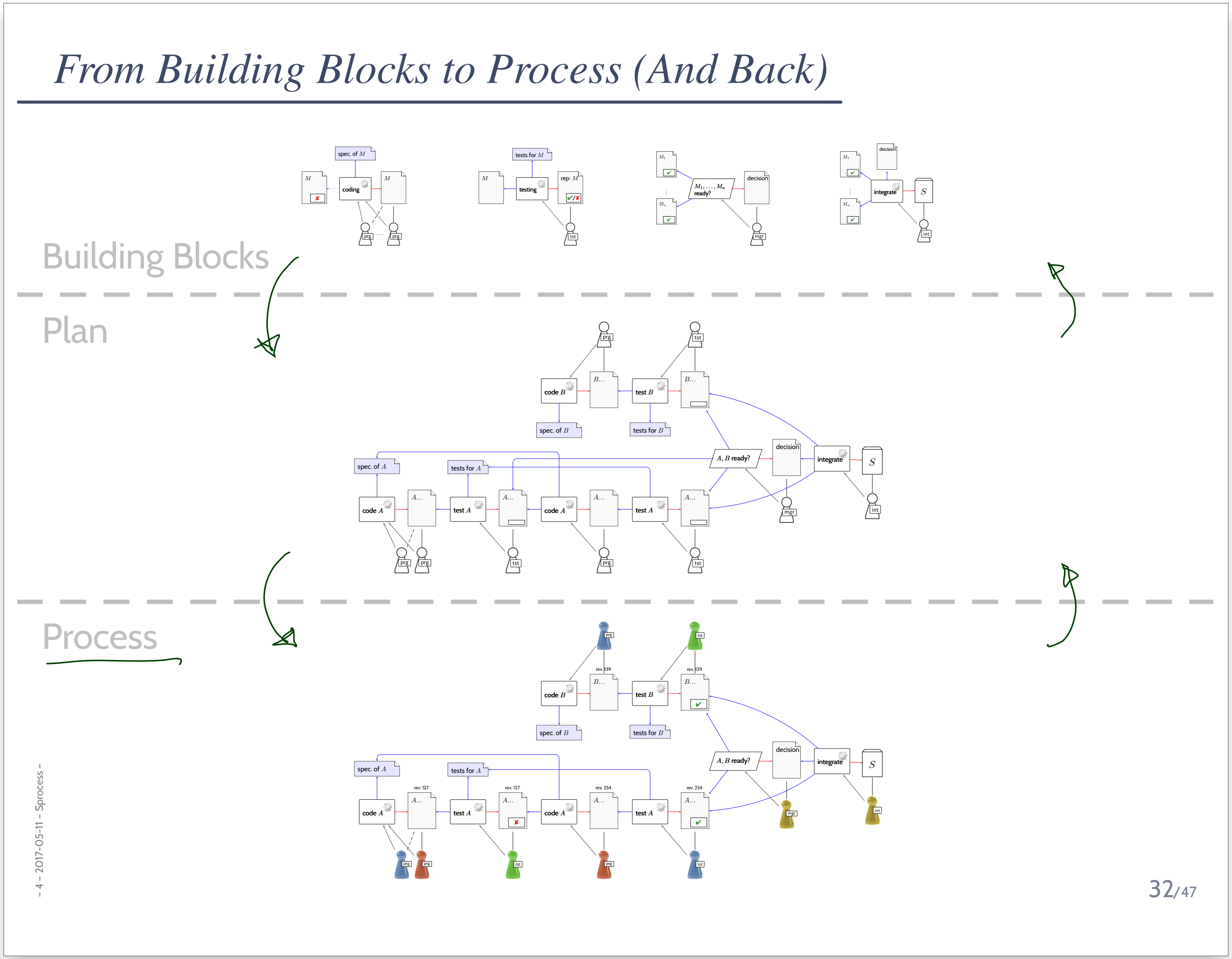
Three properties are constituent:

(i) the **image attribute** (Abbildungsmerkmal), i.e. there is an entity (called original) whose image or archetype the model is,

(ii) the **reduction attribute** (Verkürzungsmerkmal), i.e. only those attributes of the original that are relevant in the modelling context are represented,

(iii) the **pragmatic attribute**, i.e. the model is built in a specific context for a specific purpose.

Example: Process Model



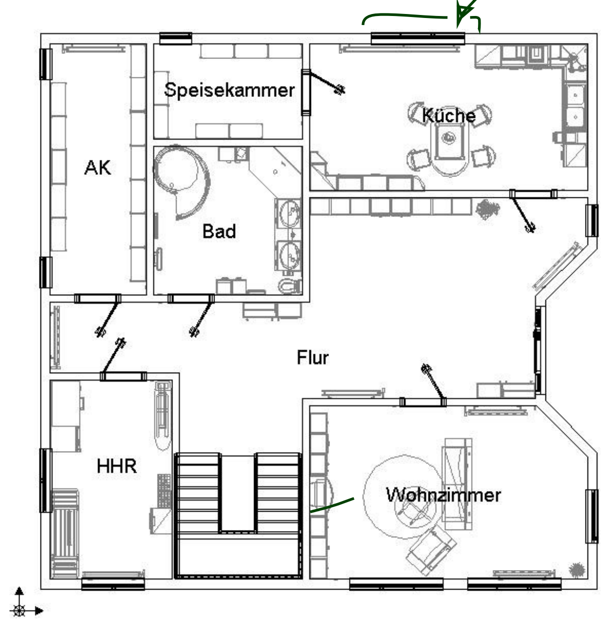
Example: Design-Models in Construction Engineering

1. Requirements

- Shall fit on given piece of land.
- Each room shall have a door.
- Furniture shall fit into living room.
- Bathroom shall have a window.
- Cost shall be in budget.

2. Designmodel

Fenster



http://wikimedia.org (CC nc-sa 3.0, Ottoklages)

3. System

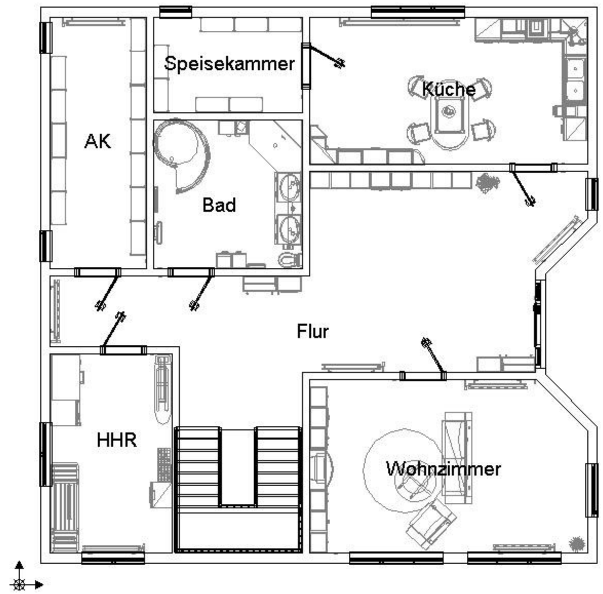
http://wikimedia.org
(CC nc-sa 3.0, Bobthebuilder82)

Example: Design-Models in Construction Engineering

1. Requirements

- Shall fit on given piece of land.
- Each room shall have a door.
- Furniture shall fit into living room.
- Bathroom shall have a window.
- Cost shall be in budget.

2. Designmodel



http://wikimedia.org (CC nc-sa 3.0, Ottoklages)

3. System

http://wikimedia.org (CC nc-sa 3.0, Bobthebuilder82)

Observation (1): Floorplan **abstracts** from certain system properties, e.g. …

- kind, number, and placement of bricks,
- subsystem details (e.g., window style),
- water pipes/wiring, and
- wall decoration

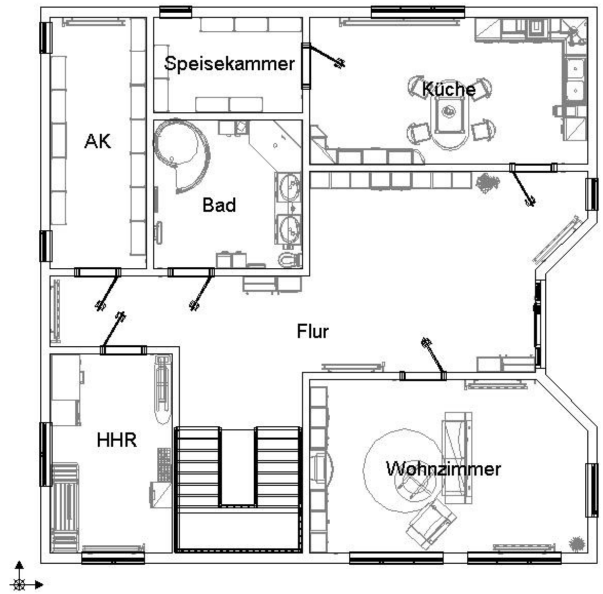
→ architects can efficiently work on appropriate level of abstraction

Example: Design-Models in Construction Engineering

1. Requirements

- Shall fit on given piece of land.
- Each room shall have a door.
- Furniture shall fit into living room.
- Bathroom shall have a window.
- Cost shall be in budget.

2. Designmodel



http://wikimedia.org (CC nc-sa 3.0, Ottoklages)

3. System

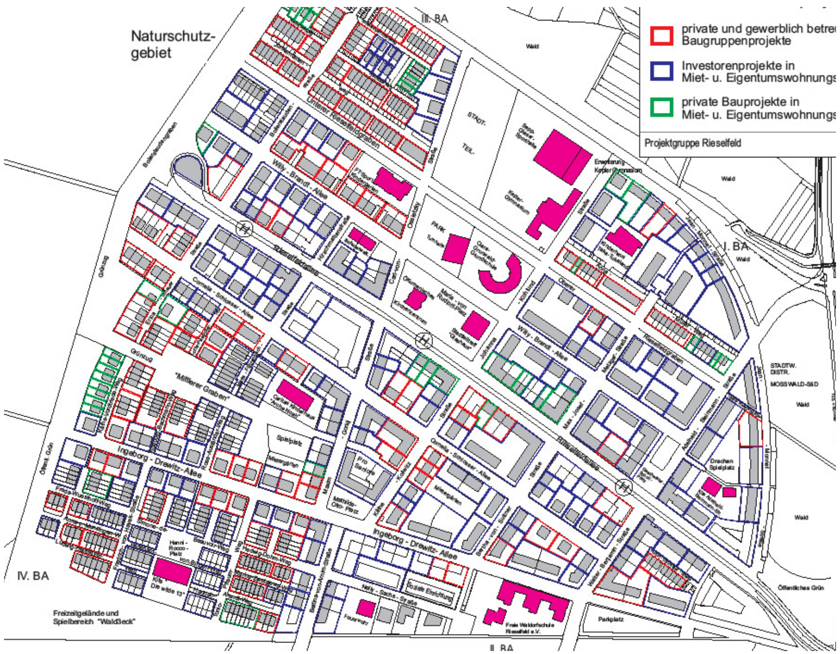
http://wikimedia.org
(CC nc-sa 3.0, Bobthebuilder82)

Observation (2): Floorplan **preserves**/**determines** certain system properties, e.g.,

- house and room extensions (to scale),
- presence/absence of windows and doors,
- placement of subsystems (such as windows).

→ find design errors before building the system (e.g. bathroom windows)

A Better Analogy is Maybe Regional Planning



(Broaddus, 2010)

http://commons.wikimedia.org/wiki/File:Freiburg_Rieselfeld_2007.jpg.
Norbert Blau, CC BY-SA 3.0

References

References

Bachmann, F., Bass, L., Clements, P., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2002). Documenting software architecture: Documenting interfaces. Technical Report 2002-TN-015, CMU/SEI.

Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. The SEI Series in Software Engineering. Addison-Wesley, 2nd edition.

Booch, G. (1993). *Object-oriented Analysis and Design with Applications*. Prentice-Hall.

Broaddus, A. (2010). A tale of two eco-suburbs in Freiburg, Germany: Parking provision and car use. *Transportation Research Record*, 2187:114–122.

Dobing, B. and Parsons, J. (2006). How UML is used. *Communications of the ACM*, 49(5):109–114.

Ellis, W. J., II, R. F. H., Saunders, T. F., Poon, P. T., Rayford, D., Sherlund, B., and Wade, R. L. (1996). Toward a recommended practice for architectural description. In *ICECCS*, pages 408–413. IEEE Computer Society.

Glinz, M. (2008). Modellierung in der Lehre an Hochschulen: Thesen und Erfahrungen. *Informatik Spektrum*, 31(5):425–434.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Harel, D. (1997). Some thoughts on statecharts, 13 years later. In Grumberg, O., editor, *CAV*, volume 1254 of *LNCS*, pages 226–231. Springer-Verlag.

Harel, D., Lachover, H., et al. (1990). Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414.

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

IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology*. Std 610.12-1990.

IEEE (2000). *Recommended Practice for Architectural Description of Software-Intensive Systems*. Std 1471.

Jacobson, I., Christerson, M., and Jonsson, P. (1992). *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley.

Kruchten, P. (1995). The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50.

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.

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

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1990). *Object-Oriented Modeling and Design*. Prentice Hall.

Rupp, C. and die SOPHISTen (2014). *Requirements-Engineering und -Management*. Hanser, 6th edition.

Taylor, R. N., Medvidovic, N., and Dahofy, E. M. (2010). *Software Architecture Foundations, Theory, and Practice*. John Wiley and Sons.