**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**

<table>
<thead>
<tr>
<th>VL 6</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Requirements Specification</td>
</tr>
<tr>
<td></td>
<td>Desired Properties</td>
</tr>
<tr>
<td></td>
<td>Kinds of Requirements</td>
</tr>
<tr>
<td></td>
<td>Analysis Techniques</td>
</tr>
<tr>
<td></td>
<td>Documents</td>
</tr>
<tr>
<td></td>
<td>Dictionary, Specification</td>
</tr>
<tr>
<td></td>
<td>Specification Languages</td>
</tr>
<tr>
<td></td>
<td>Natural Language</td>
</tr>
<tr>
<td></td>
<td>Decision Tables</td>
</tr>
<tr>
<td></td>
<td>Syntax, Semantics</td>
</tr>
<tr>
<td></td>
<td>Completeness, Consistency,...</td>
</tr>
<tr>
<td>VL 7</td>
<td>Scenarios</td>
</tr>
<tr>
<td></td>
<td>User Stories, Use Cases</td>
</tr>
<tr>
<td></td>
<td>Live Sequence Charts</td>
</tr>
<tr>
<td></td>
<td>Syntax, Semantics</td>
</tr>
<tr>
<td>VL 8</td>
<td>Definition: Software &amp; SW Specification</td>
</tr>
<tr>
<td>VL 9</td>
<td>Wrap-Up</td>
</tr>
<tr>
<td>VL 10</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
TBA Construction Principle

"Only" construct the transitions' labels:

\[ \Rightarrow \{ (q, \psi_{\text{loop}}(q), q) \mid q \in Q \} \cup \{ (q, \psi_{\text{prog}}(q, q'), q') \mid q \rightarrow q' \} \cup \{ (q, \psi_{\text{exit}}(q), L) \mid q \in Q \} \]

\[ \psi_{\text{loop}}(q) = \psi_{\text{hot}}(q) \wedge \psi_{\text{cold}}(q) \]

\[ \psi_{\text{prog}}(q, q_n) = \psi_{\text{hot}}(q, q_n) \wedge \psi_{\text{cold}}(q, q_n) \]

\[ \psi_{\text{exit}}(q) = \psi_{\text{cold}}(q) \]

[Diagram of TBA construction principle with states and transitions labeled with formulas.]
**Loop Condition**

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

- \( \psi_{\text{Msg}}(q) = \neg \bigwedge_{1 \leq i \leq n} \psi_{\text{Msg}}(q_i, q_i) \land (\text{strict} \implies \bigwedge_{q \in E} \neg \psi) \)
- \( \psi_{\text{LocInv}}(q) = \bigwedge_{(i, \phi, l', \theta') \in \text{LocInv}, \theta'(l') = \theta} \exists \text{ active at } q \phi \)

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

Local invariant \( (l_0, t_0, \phi, l_1, l_2) \) is active at cut \( l \) if and only if \( l_0 \leq l < l_1 \) for some front location \( l \) of cut \( q \) or \( l = l_1 \).

- \( \text{Msg}(\mathcal{F}) = \{ \text{Msg}(l) \mid \text{Msg} \in \mathcal{F} \} \cup \{ E \mid (l, E, l') \in \text{Msg}, l' \in \mathcal{F} \} \)
- \( \text{Msg}(\mathcal{F}_1, \ldots, \mathcal{F}_m) = \bigcup_{1 \leq i \leq n} \text{Msg}(\mathcal{F}_i) \)

**Progress Condition**

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

- \( \psi_{\text{Msg}}(q, q_i) = \bigwedge_{q \in \text{Msg}(q, q_i)} \psi \land \bigwedge_{q \neq \psi} \bigwedge_{q \in \text{Msg}(q, q_i)} \neg \psi \)
- \( \psi_{\text{LocInv}}(q, q_i) = \bigwedge_{(i, \phi, l', \theta') \in \text{LocInv}, \theta'(l') = \theta} \exists \text{ active at } q \phi \)

Local invariant \( (l_0, t_0, \phi, l_1, l_2) \) is **active at** \( q \) if and only if

- \( l_0 < l < l_1 \), or
- \( l = l_1 \) and \( l_2 = \bullet \), or
- \( l = l_1 \) and \( l_2 = \bullet \)

for some front location \( l \) of cut \( l \). q.
Content

- LSCs: Automaton Construction
- Excursion: Symbolic Büchi Automata
- LSCs vs. Software
- Methodology
  - Requirements Engineering with scenarios
  - Strengthening scenarios 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

\( A : \Sigma = \{0, 1\} \)

\( A_{\text{sym}} : \Sigma = (\{x\} \rightarrow \mathbb{N}) \)

\( B : \Sigma = \{0, 1\} \)

\( B_{\text{sym}} : \Sigma = (\{x\} \rightarrow \mathbb{N}) \)

\( L(A) = \{0 (1, 0)^* \} \)

\( w = 0101010 \ldots \in L(A) \)

\( w = 0101010 \ldots \in L(B) \)

\( w_{A} \in (10)^* \in L(A_{\text{sym}}) \)

\( w_{B} \in (10)^* \in L(B_{\text{sym}}) \)

\( L(B_{\text{sym}}) = (10)^* \in L(A_{\text{sym}}) \)

\( L(B_{\text{sym}}) = (10)^* \in L(A_{\text{sym}}) \)

\( w \in (10)^* \in L(A_{\text{sym}}) \)

\( w \in (10)^* \in L(B_{\text{sym}}) \)

\( \omega = (x, 1, 0) \in L(A) \)

\( \omega = (x, 1, 0) \in L(B) \)

\( \omega = (x, 1, 0) \in L(A_{\text{sym}}) \)

\( \omega = (x, 1, 0) \in L(B_{\text{sym}}) \)
Definition. A **Symbolic Büchi Automaton** (TBA) is a tuple

$$B = (C_B, Q, q_{ini}, \rightarrow, Q_F)$$

where

- $C_B$ is a set of atomic propositions,
- $Q$ is a finite set of states,
- $q_{ini} \in Q$ is the initial state,
- $\rightarrow \subseteq Q \times \Phi(C_B) \times Q$ is the finite transition relation.

Each transition $(q, \psi, q') \in \rightarrow$ from state $q$ to state $q'$ is labelled with a formula $\psi \in \Phi(C_B)$.

- $Q_F \subseteq Q$ is the set of fair (or accepting) states.

### Run of TBA

Definition. Let $B = (C_B, Q, q_{ini}, \rightarrow, Q_F)$ be a TBA and

$$w = \sigma_1, \sigma_2, \sigma_3, \ldots \in (\Phi(C_B) \rightarrow B)^\omega$$

an infinite word, each letter is a valuation of $\Phi(C_B)$.

An infinite sequence

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

of states is called **run** of $B$ over $w$ if and only if

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

**Example:**

$$B_{sym}: \text{even}(x) \quad \Sigma = \{\{x\} \rightarrow \mathbb{N}\}$$
Definition.
We say TBA $B = (C_B, Q, q_in, \rightarrow, Q_F)$ accepts the word
$$w = (\sigma_i)_{i \in \mathbb{N}_0} \in (\Phi(C_B) \rightarrow B)\omega$$
if and only if $B$ has a run $\varrho = (q_i)_{i \in \mathbb{N}_0}$
over $w$ such that fair (or accepting) states are visited infinitely often
by $\varrho$, i.e., such that
$$\forall i \in \mathbb{N}_0 \exists j > i : q_j \in Q_F.$$

We call the set $Lang(B) \subseteq (\Phi(C_B) \rightarrow B)\omega$ of words that are accepted by $B$
the language of $B$.

Example:

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

A computation path $\pi = \sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \in [S]$ 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 say software $S$ satisfies LSC $L$ (without pre-chart), denoted by $S \models L$, if and only if
\[
\begin{array}{c|c|c}
\text{cold} & \text{am = initial} & \text{am = invariant} \\
\hline
\exists w \in W_S \cdot w^0 \models ac \land \neg \psi_{exit}(C_0) \\
\land w^0 \models \psi_{prog}(\emptyset, C_0) \land w/1 \in \text{Lang}(B(L)) & \exists w \in W_S \exists k \in \mathbb{N}_0 \cdot w^k \models ac \land \neg \psi_{exit}(C_0) \\
\land w^k \models \psi_{prog}(\emptyset, C_0) \land w/k + 1 \in \text{Lang}(B(L)) & \\
\hline
\end{array}
\]

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

---

**Example: TBA vs. Computation Path**
A full LSC $\mathcal{L} = (PC, MC, ac, am, \Theta_{\mathcal{L}})$ actually consists of

- **pre-chart** $PC = ((\mathcal{L}_P, \leq_P, \sim_P), \mathcal{I}_P, \mathcal{M}_P, \mathcal{D}_P, \mathcal{L}(\mathcal{P}_P), \Theta_P)$ (possibly empty),
- **main-chart** $MC = ((\mathcal{L}_M, \leq_M, \sim_M), \mathcal{I}_M, \mathcal{M}_M, \mathcal{D}_M, \mathcal{L}(\mathcal{P}_M), \Theta_M),$
- activation condition $ac \in \Phi(C),$ and mode $am \in \{\text{initial, invariant}\},$
- strictness flag $\Theta_{\mathcal{L}} = \text{strict}$ or $\Theta_{\mathcal{L}} = \text{universal}$ ($\Theta_{\mathcal{L}} = \text{hot}$).

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

For $am = \text{initial}$

- $\exists w \in W \exists m \in \mathbb{N}_0 \bullet \Theta_{\mathcal{L}}$ such that
  - $w^0 \models ac \land \neg \psi_{\text{exit}}(C_{0}^{\mathcal{P}}) \land \psi_{\text{prog}}(\theta, C_{0}^{\mathcal{M}})$
  - $w_1, \ldots, w/m \in \text{Lang}(B(\mathcal{P}(C)))$
  - $w^{m+1}_1 \models \neg \psi_{\text{exit}}(C_{M}^{\mathcal{P}})$
  - $w/m + 2 \in \text{Lang}(B(\mathcal{M}(C)))$

For $am = \text{invariant}$

- $\forall w \in W \forall m \in \mathbb{N}_0 \bullet \Theta_{\mathcal{L}}$ such that
  - $w^0 \models ac \land \neg \psi_{\text{exit}}(C_{0}^{\mathcal{P}}) \land \psi_{\text{prog}}(\theta, C_{0}^{\mathcal{M}})$
  - $w_1, \ldots, w/m \in \text{Lang}(B(\mathcal{P}(C)))$
  - $w^{m+1}_1 \models \neg \psi_{\text{exit}}(C_{M}^{\mathcal{P}})$
  - $w/m + 2 \in \text{Lang}(B(\mathcal{M}(C)))$

where $C_{0}^{\mathcal{P}}$ and $C_{0}^{\mathcal{M}}$ 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 $\mathcal{L}$ 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)
How to Prove that a Software Satisfies an LSC?

- Software $S$ satisfies **existential** LSC $\mathcal{L}$ if there exists $\pi \in \llbracket S \rrbracket$ 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.)

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

Pushing Things Even Further

(Harel and Marelly, 2003)
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**

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

![Diagram showing a system with buttons, sensors, motors, and controllers.](image-url)
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**
  - Each requirement is compatible with all other requirements; otherwise the requirements are not realizable.
- **Neutral/General**
  - 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.

- e.g. 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 \( \bigwedge_{i=1}^{n} W \models L_i \).
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
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,
- $MC$ – dispense cash and return card.

- Customer 1: “don’t care”
  \[ \mathcal{S}_1 = (M.C|M.C|M.C) \]

- 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
Topic Area Architecture & Design: Content

- **Introduction and Vocabulary**

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

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

- **Principles of Design**
  a) modularity
  b) separation of concerns
  c) information hiding and data encapsulation
  d) abstract data types, object orientation

- **Modelling behaviour**
  a) communicating finite automata
  b) Uppaal query language
  c) basic state-machines
  d) an outlook on hierarchical state-machines

- **Design Patterns**
Content

- LSCs: Automaton Construction
- Excursion: Symbolic Büchi Automata
- LSCs vs. Software
- Methodology
  - Requirements Engineering with scenarios
  - Strengthening scenarios 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
**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**
**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 insofar as explicitly permitted by the programmers. (i.e.: *visible from the outside*)  
**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**
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)
Goal 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":

**Content**

- LSCs: **Automaton Construction**
- Excursion: **Symbolic Büchi Automata**
- LSCs vs. **Software**
- **Methodology**
  - Requirements Engineering with scenarios
  - Strengthening scenarios 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
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**

*From Building Blocks to Process (And Back)*

**Building Blocks**

**Plan**

**Process**

---

**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

3. System
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

3. System

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

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

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

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


