### Topic Area Architecture & Design: Content

| VL 11 | • Introduction and Vocabulary  
|       | • Software Modelling  
|       |   • model, views / viewpoints: 4+1 view  
|       | • Modelling structure  
| VL 12 | • (simplified) class & object diagrams  
|       | • (simplified) object constraint logic (OCL)  
| VL 13 | • Principles of Design  
|       |   • modularity, separation of concerns  
|       |   • information hiding and data encapsulation  
|       |   • abstract data types, object orientation  
|       | • Design Patterns  
| VL 14 | • Modelling behaviour  
|       |   • communicating finite automata (CFA)  
|       |   • Uppaal query language  
| VL 15 | • CFA vs. Software  
|       | • Model-driven/-based Software Engineering  
|       | • Unified Modelling Language (UML)  
|       |   • basic and hierarchical state-machines  

---
CFA vs. Software

- a CFA model is software

CFA at Work

- drive to configuration, scenarios, invariants
- tool demo (verifier).

Model-based/-driven Software Engineering

Unified Modelling Language

- Brief History
- Sub-Languages
- UML Modes

UML State Machines

- Hierarchical State Machines
- Core State Machines
- steps and run-to-completion steps
- Rhapsody

CFA vs. 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
\[ \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \]
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 $S$.

- Let $C(A_1, \ldots, A_n)$ be a network of CFA.
- $\Sigma = \text{Conf}$
- $A = \text{Act}$
- $\llbracket C \rrbracket = \{ \pi = (\vec{l}_0, \nu_0) \xrightarrow{\lambda_1} (\vec{l}_1, \nu_1) \xrightarrow{\lambda_2} (\vec{l}_2, \nu_2) \xrightarrow{\lambda_3} \cdots | \pi \text{ is a computation path of } C \}$.
- Note: the structural model just consists of the set of variables and the locations of $C$.

Example: Software Specification

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

- **Customer:** "I don’t care about the order of $M$ and $C$"
  \[ \mathcal{S}_1 = (M.C | C.M | M/C)^\omega \]

- **Refined Specification:** "be consistent: either always $M.C$ or always $C.M$"
  \[ \mathcal{S}_2 = (M.C)^\omega \text{ or } (C.M)^\omega \]

- **Design Idea:** "consider human errors: always do $C.M$"
  \[ S_1 = (C.M)^\omega \]

- **Implementation** (goal): software $S_2$ behaves according to the design idea.
CFA and Queries at Work
Example: Vending Machine — Model Architecture

- **Shared variables:**
  - bool water_enabled, soft_enabled, tea_enabled;
  - int \( w = 3, s = 3, t = 3 \);

- **Note:** Our model does not use scopes ("information hiding") for channels. That is, 'Service' could send 'WATER' if the modeler wanted to.

**Design Sanity Check: Drive to Configuration**

- **Question:** Is it (at all) possible to have no water in the vending machine model? (Otherwise, the design is definitely broken.)

- **Approach:** Check whether a configuration satisfying

  \[ w = 0 \]

  is reachable, i.e. check

  \[ \mathcal{N}_{VM} \models \exists w. w = 0. \]

  for the vending machine model, \( \mathcal{N}_{VM} \).
**Design Check: Scenarios**

- **Question:** Is the following existential LSC satisfied by the model? (Otherwise, the design is definitely broken.)

  LSC: buy tea

  AC: true

  AM: initial I: permissive

  User

  Coin Validator

  Choice Panel

  C

  50

  C

  50

  C

  50

  TEA

  ¬

  E1

- **Approach:** Use the following newly created CFA 'Scenario'

  instead of **User** and check whether location **end_of_scenario** is reachable, i.e. check

  \[ \mathcal{N}_VM \models \exists \text{Scenario.end_of_scenario.} \]

  for the modified vending machine model \( \mathcal{N}_VM \).

**Design Verification: Invariants**

- **Question:** Is it the case that the “tea” button is only enabled if there is €1.50 in the machine? (Otherwise, the design is broken.)

- **Approach:** Check whether the implication

  \[ \text{tea_enabled} \Rightarrow \text{CoinValidator.have_c150} \]

  holds in all reachable configurations, i.e. check

  \[ \mathcal{N}_VM \models \forall \square \text{tea_enabled imply CoinValidator.have_c150} \]

  for the vending machine model \( \mathcal{N}_VM \).
**Question:** Is the “tea” button ever enabled?

(If otherwise, the considered invariant

\[
\text{tea\_enabled} \implies \text{CoinValidator\_have\_c150}
\]

holds vacuously.)

**Approach:** Check whether a configuration satisfying \(\text{water\_enabled} = 1\) is reachable.

Exactly like we did with \(w = 0\) earlier.

---

**Design Verification: Another Invariant**

**Question:** Is it the case that, if there is money in the machine and water in stock, that the “water” button is enabled?

**Approach:** Check

\[\mathcal{N}_{\text{VM}} \models \forall (\text{CoinValidator\_have\_c50} \lor \text{CoinValidator\_have\_c100} \lor \text{CoinValidator\_have\_c150}) \implies \text{water\_enabled}.\]
Recall: Universal LSC Example

- LSC: buy water
  - AC: true
  - AM: invariant I: strict

User
CoinValidator
ChoicePanel
Dispenser

C
50
pWATER

¬(C50 ! ∨ pSOFT ! ∨ pTEA ! ∨ pFILLUP !)

water
in
stock
dWATER
OK
¬(dSoft ! ∨ dTEA !)

Content I (Architecture & Design)

- CFA vs. Software
  - a CFA model is software

- CFA at Work
  - drive to configuration, scenarios, invariants
  - tool demo (verifier).

- Model-based/-driven Software Engineering

  - Unified Modelling Language
    - Brief History
    - Sub-Languages
    - UML Modes

- UML State Machines
  - Hierarchical State Machines
  - Core State Machines
  - steps and run-to-completion steps
  - Rhapsody
Model-based/-driven Software Engineering
• (Jacobson et al., 1992): “System development is model building.”
• Model-based software engineering (MBSE): some formal models are used.
• Model-driven software engineering (MDSE): all artefacts are (formal) models.
**Development Approaches**

- **top-down** risk: needed functionality hard to realise on target platform.
- **bottom-up** risk: lower-level units do not “fit together”.
- **inside-out** risk: user interface needed by customer hard to realise with existing system.
- **outside-in** risk: elegant system design not reflected nicely in (already fixed) UI.

**Transform vs. Write-Down-and-Check**

**Software Modelling**

- Constructive
- Reflective
int w := 3;
typedef {Wi, dispense, W0} st_T;
st_T st := W1;
Set(Act) take_action(Act α) {
Set(Act) R := ∅;
if □ st = Wi :
  if □ α = DWATER? : w := w - 1;
    st := dispense;
    if (w = 0) R := R ∪ {DOK!};
    if (w > 0) R := R ∪ {DOK!};
  □ α = FILLUP? :
    w := 3;
    st := W1;
    R := R ∪ {DWATER?, FILLUP?};
  □ else:
    R := R ∪ {DWATER?, FILLUP?};
fi
□ st = dispense :
  if □ α = DOK! ∧ w = 0 : st := W0;
    R := R ∪ {FILLUP?};
  □ α = DOK! ∧ w > 0 : st := W1;
    R := R ∪ {FILLUP?};
  □ else:
    R := R ∪ {DOK?};
fi
□ st = W0 :
  if □ α = FILLUP? :
    w := 3;
    st := W1;
    R := R ∪ {DWATER?, FILLUP?};
  □ else:
    R := R ∪ {FILLUP?};
fi
□: return R;
}

Content I (Architecture & Design)

- CFA vs. Software
  - a CFA model is software

- CFA at Work
  - drive to configuration, scenarios, invariants
  - tool demo (verifier).

- Model-based/-driven Software Engineering

  - Unified Modelling Language
    - Brief History
    - Sub-Languages
    - UML Modes

- UML State Machines
  - Hierarchical State Machines
  - Core State Machines
  - steps and run-to-completion steps
  - Rhapsody
**A Brief History of the Unified Modelling Language (UML)**

- Boxes/lines and automata are used to visualise software for ages.
- 1970’s, **Software Crisis™** – Idea: learn from engineering disciplines in order to handle growing complexity.
  - Modelling languages: Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams
- Mid 1980’s: **Statecharts** (Harel, 1987), **StateMate™** (Harel et al., 1990)
- Early 1990’s, advent of **Object-Oriented Analysis/Design/Programming** – Inflation of notations and methods, most prominent:
  - **Object-Modeling Technique** (OMT) (Rumbaugh et al., 1990)
  - **Booch Method and Notation** (Booch, 1993)
  - **Object-Oriented Software Engineering** (OOSE) (Jacobson et al., 1992)
  - Each “persuasion” selling books, tools, seminars...
- Late 1990’s: joint effort of “the three amigos” UML 0.x and 1.x
  - Standards published by **Object Management Group** (OMG):
    - “International, open membership, not-for-profit computer industry consortium”
  - Much criticised for lack of formality.
- Since 2005: **UML 2.x**, split into infra- and superstructure documents.
  - Syntax: pretty precisely defined.
  - Semantics: natural language, thus informal.
**UML Overview** (OMG, 2007b, 684)

**Figure A.5 - The taxonomy of structure and behavior diagram**

Recall: definition “model” (Glinz, 2008, 425):

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

Examples for context/purpose:

**Floorplan as sketch:**

![Floorplan as sketch]

**Floorplan as blueprint:**

![Floorplan as blueprint]

**Floorplan as program:**

![Floorplan as program]
The last slide is inspired by Martin Fowler, who puts it like this:

“[..] people differ about what should be in the UML because there are differing fundamental views about what the UML should be.

I came up with three primary classifications for thinking about the UML: 
UmlAsSketch, UmlAsBlueprint, and UmlAsProgrammingLanguage. 
([..] S. Mellor independently came up with the same classifications.)

So when someone else’s view of the UML seems rather different to yours, it may be because they use a different UmlMode to you.”

Claim:
• This not only applies to UML as a language (what should be in it etc.?),
• but at least as well to each individual UML model.

UML-Mode of the Lecture: As Blueprint

Sketch
In this UmlMode developers use the UML to help communicate some aspects of a system. [...] Sketches are also useful in documents, in which case the focus is communication rather than completeness. [...] The tools used for sketching are lightweight drawing tools and often people aren’t too particular about keeping to every strict rule of the UML. Most UML diagrams shown in books, such as mine, are sketches. Their emphasis is on selective communication rather than complete specification.

Hence my sound-bite “comprehensiveness is the enemy of comprehensibility”

Blueprint
[..] In forward engineering the idea is that blueprints are developed by a designer whose job is to build a detailed design for a programmer to code up. That design should be sufficiently complete that all design decisions are laid out and the programming should follow as a pretty straightforward activity that requires little thought. [...] Blueprints require much more sophisticated tools than sketches in order to handle the details required for the task. [...] Forward engineering tools support diagram drawing and back it up with a repository to hold the information. [...]”

ProgrammingLanguage
If you can detail the UML enough, and provide semantics for everything you need in software, you can make the UML be your programming language. Tools can take the UML diagrams you draw and compile them into executable code.

The promise of this is that UML is a higher level language and thus more productive than current programming languages.

The question, of course, is whether this promise is true. I don’t believe that graphical programming will succeed just because it’s graphical. [..]

Our goal:
• be precise to avoid misunderstandings.
• allow formal analysis of consistency/implication on the design level — find errors early.

Yet we tried to be consistent with the (informal semantics) from OMG (2007a,b) as far as possible.
Composite (or Hierarchical) States

- Composite states are about abbreviation, structuring, and avoiding redundancy.
Example

Would be Too Easy...

→ "Software Design, Modelling, and Analysis with UML" in some winter semesters.
```plaintext
annot ::= [⟨event⟩, ⟨event⟩]*
       [⟨guard⟩] [⟨action⟩]

with
* event ∈ E,
* guard ∈ Expr S (default: true, assumed to be in Expr S)
* action ∈ Act S (default: skip, assumed to be in Act S)

(Event Pool and Run-To-Completion)

step  | state | stable | x  | state | stable | event pool
------|-------|--------|----|-------|--------|-----------------
Event Pool and Run-To-Completion

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
\text{step} & \text{state} & \text{stable} & x & \text{state} & \text{stable} & \text{event pool} \\
\hline
0 & s_1 & 1 & 27 & s_1 & 1 & E \text{ ready for } u_1 \\
1 & s_2 & 1 & 27 & s_1 & 1 & F \text{ ready for } u_2 \\
\hline
\end{array}
\]

Event Pool and Run-To-Completion

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
\text{step} & \text{state} & \text{stable} & x & \text{state} & \text{stable} & \text{event pool} \\
\hline
0 & s_1 & 1 & 27 & s_1 & 1 & E \text{ ready for } u_1 \\
1 & s_2 & 1 & 27 & s_1 & 1 & F \text{ ready for } u_2 \\
\hline
\end{array}
\]
Event Pool and Run-To-Completion

\[
\begin{array}{cccc|c}
\text{state} & \text{stable} & \text{state} & \text{stable} & \text{event pool} \\
\hline
s_1 & 1 & s_1 & 1 & E \text{ ready for } u_1 \\
2 & s_2 & 1 & s_1 & 1 & F \text{ ready for } u_2 \\
3 & s_2 & 1 & s_2 & 0 & G \text{ ready for } u_1 \\
\end{array}
\]

\[
\begin{array}{cccc|c}
\text{state} & \text{stable} & \text{state} & \text{stable} & \text{event pool} \\
\hline
s_1 & 1 & s_1 & 1 & E \text{ ready for } u_1 \\
2 & s_2 & 1 & s_1 & 1 & F \text{ ready for } u_2 \\
3 & s_2 & 1 & s_2 & 0 & G \text{ ready for } u_1 \\
4.a & s_2 & 1 & s_1 & 1 & G \text{ ready for } u_1 \\
\end{array}
\]
### Event Pool and Run-To-Completion

**Table:**

<table>
<thead>
<tr>
<th>Step</th>
<th>State</th>
<th>Stable</th>
<th>$x$</th>
<th>State</th>
<th>Stable</th>
<th>Event Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$s_1$</td>
<td>1</td>
<td>$s_2$</td>
<td>1</td>
<td></td>
<td>$E$ ready for $u_1$</td>
</tr>
<tr>
<td>1</td>
<td>$s_2$</td>
<td>1</td>
<td>$s_1$</td>
<td>1</td>
<td></td>
<td>$F$ ready for $u_2$</td>
</tr>
<tr>
<td>2</td>
<td>$s_2$</td>
<td>1</td>
<td>$s_2$</td>
<td>0</td>
<td></td>
<td>$G$ ready for $u_1$</td>
</tr>
<tr>
<td>3</td>
<td>$s_2$</td>
<td>1</td>
<td>$s_3$</td>
<td>0</td>
<td></td>
<td>$G$ ready for $u_1$</td>
</tr>
<tr>
<td>4a</td>
<td>$s_2$</td>
<td>1</td>
<td>0</td>
<td>$s_1$</td>
<td>1</td>
<td>$G$ ready for $u_1$</td>
</tr>
<tr>
<td>5a</td>
<td>$s_1$</td>
<td>1</td>
<td>0</td>
<td>$s_1$</td>
<td>1</td>
<td>$G$ ready for $u_1$</td>
</tr>
</tbody>
</table>

**Diagram:**

- $u_1 : C$
  - state: $\{s_1, s_2\}$
  - stable: Bool

- $u_2 : D$
  - state: $\{s_1, s_2, s_3\}$
  - stable: Bool

- Event pool:
  - $0$ seconds
  - $1$ second
  - $2$ seconds
  - $3$ seconds
### Event Pool and Run-To-Completion

<table>
<thead>
<tr>
<th>step</th>
<th>state $u_1$ stable</th>
<th>$x$</th>
<th>state $u_2$ stable</th>
<th>event pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s1 1</td>
<td>27</td>
<td>s1 1</td>
<td>E ready for $u_1$</td>
</tr>
<tr>
<td>1</td>
<td>s2 1</td>
<td>27</td>
<td>s1 1</td>
<td>F ready for $u_2$</td>
</tr>
<tr>
<td>2</td>
<td>s2 1</td>
<td>27</td>
<td>s2 0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3 1</td>
<td>27</td>
<td>s3 0</td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>s2 1</td>
<td>0</td>
<td>s1 1</td>
<td>G ready for $u_1$</td>
</tr>
<tr>
<td>5a</td>
<td>s1 1</td>
<td>0</td>
<td>s1 1</td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>s1 1</td>
<td>27</td>
<td>s3 0</td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>s1 1</td>
<td>0</td>
<td>s1 1</td>
<td></td>
</tr>
</tbody>
</table>

---

### Rhapsody Architecture

- **Generate**
  - C.h
  - D.h
  - C.cpp
  - D.cpp
  - MainDefaultComponent.cpp
  - DfltCmp.exe

- **Run**
  - build / make (compiler)

- **Event**
  - $E! \rightarrow G$
  - $D$ just stepped from $s_1$ to $s_2$ by transition $r$
Tell Them What You’ve Told Them...

- We can use tools like Uppaal to
  - check and verify CFA design models against requirements.

- CFA (and state charts)
  - can easily be implemented using the translation scheme.

- Wanted: verification results carry over to the implementation.
  - if code is not generated automatically, verify code against model.

- UML State Machines are
  - principally the same thing as CFA, yet provide more convenient syntax.
  - Semantics uses
    - asynchronous communication,
    - run-to-completion steps
  - in contrast to CFA.
  (We could define the same for CFA, but then the Uppaal simulator would not be useful any more.)
# Topic Area Code Quality Assurance: Content

<table>
<thead>
<tr>
<th>VL 15</th>
<th>Introduction and Vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Test case, test suite, test execution.</td>
</tr>
<tr>
<td></td>
<td>Positive and negative outcomes.</td>
</tr>
<tr>
<td></td>
<td>Limits of Software Testing</td>
</tr>
<tr>
<td>VL 16</td>
<td>Glass-Box Testing</td>
</tr>
<tr>
<td></td>
<td>Statement-, branch-, term-coverage.</td>
</tr>
<tr>
<td></td>
<td>Other Approaches</td>
</tr>
<tr>
<td></td>
<td>Model-based testing,</td>
</tr>
<tr>
<td></td>
<td>Runtime verification.</td>
</tr>
<tr>
<td>VL 17</td>
<td>Program Verification</td>
</tr>
<tr>
<td></td>
<td>partial and total correctness,</td>
</tr>
<tr>
<td></td>
<td>Proof System PD.</td>
</tr>
<tr>
<td>VL 18</td>
<td>Review</td>
</tr>
</tbody>
</table>
Testing: Introduction
Quotes On Testing

"Testing is the execution of a program with the goal to discover errors."  
(G. J. Myers, 1979)

"Testing is the demonstration of a program or system with the goal to show that it does what it is supposed to do."  
(W. Hetzel, 1984)

"Software testing can be used to show the presence of bugs, but never to show their absence!"  
(E. W. Dijkstra, 1970)

Rule-of-thumb: (fairly systematic) tests discover half of all errors.  
(Ludewig and Lichter, 2013)

Preliminaries

Recall:

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

The (possibly partial) function $\llbracket \cdot \rrbracket : S \rightarrow \llbracket S \rrbracket$ is called interpretation of $S$.

- From now on, we assume that states consist of an input and an output/internal part, i.e., there are $\Sigma_{\text{in}}$ and $\Sigma_{\text{out}}$ such that $\Sigma = \Sigma_{\text{in}} \times \Sigma_{\text{out}}$.
- Computation paths are then of the form
  $$\pi = \left( \begin{array}{c} \sigma_0 \\ \sigma_0' \end{array} \right) \xrightarrow{\alpha_1} \left( \begin{array}{c} \sigma_1 \\ \sigma_1' \end{array} \right) \xrightarrow{\alpha_2} \cdots$$
- We use $\pi \downarrow \Sigma_{\text{in}}$ to denote $\pi = \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \cdots$, i.e. the projection of $\pi$ onto $\Sigma_{\text{in}}$. 
Test Case

Definition. A test case $T$ over $\Sigma$ and $A$ is a pair $(In, Soll)$ consisting of
- a description $In$ of sets of finite input sequences,
- a description $Soll$ of expected outcomes,
and an interpretation $\llbracket \cdot \rrbracket$ of these descriptions:
- $\llbracket In \rrbracket \subseteq (\Sigma_{in} \times A)^*$,  \llbracket Soll \rrbracket \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$

Examples:
- Test case for procedure $\text{strlen} : \text{String} \to \mathbb{N}$, $s$ denotes parameter, $r$ return value:
  \[ T = (s = "abc", r = 3) \]
  \[ \llbracket s = "abc" \rrbracket = \{ \sigma_i \xrightarrow{\tau} \sigma_{i+1} \mid \sigma_i(s) = "abc" \}, \quad \llbracket r = 3 \rrbracket = \{ \sigma_0 \xrightarrow{\tau} \sigma_1 \mid \sigma_1(r) = 3 \}. \]
  Shorthand notation: $T = ("abc", 3)$.
- “Call $\text{strlen}()$ with string "abc", expect return value 3.”

Test Case

Definition. A test case $T$ over $\Sigma$ and $A$ is a pair $(In, Soll)$ consisting of
- a description $In$ of sets of finite input sequences,
- a description $Soll$ of expected outcomes,
and an interpretation $\llbracket \cdot \rrbracket$ of these descriptions:
- $\llbracket In \rrbracket \subseteq (\Sigma_{in} \times A)^*$,  \llbracket Soll \rrbracket \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$

Examples:
- Test case for vending machine.
  \[ T = (C50, \text{WATER}, \text{DWATER}) \]
  \[ \llbracket C50, \text{WATER} \rrbracket = \{ \sigma_0 \xrightarrow{C50} \sigma_1 \xrightarrow{\text{WATER}} \cdots \xrightarrow{\text{WATER}} \sigma_{j-1} \xrightarrow{\text{WATER}} \sigma_j \}, \]
  \[ \llbracket \text{DWATER} \rrbracket = \{ \sigma_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} \sigma_{k-1} \xrightarrow{\text{DWATER}} \sigma_k \mid k \leq 10 \}. \]
- “Send event C50 and any time later WATER, expect DWATER after 10 steps the latest.”
**Definition.** A test case $T$ over $\Sigma$ and $A$ is a pair $(I_n, S_{oll})$ consisting of

- a description $I_n$ of sets of finite input sequences,
- a description $S_{oll}$ of expected outcomes,

and an interpretation $\llbracket \cdot \rrbracket$ of these descriptions:

- $\llbracket I_n \rrbracket \subseteq (\Sigma_{in} \times A)^*$,
- $\llbracket S_{oll} \rrbracket \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$.

**Note:**

- **Input sequences** can consider
  - input data, possibly with timing constraints,
  - other interaction, e.g., from network,
  - initial memory content,
  - etc.

- **Input sequences** may leave degrees of freedom to tester.

- **Expected outcomes** may leave degrees of freedom to system.

---

**Executing Test Cases**

- A computation path

  $\pi = \left( \begin{array}{c} \sigma_0^0 \\ \sigma_0^0 \end{array} \right) \xrightarrow{\alpha_1} \left( \begin{array}{c} \sigma_1^1 \\ \sigma_1^1 \end{array} \right) \xrightarrow{\alpha_2} \ldots$

  from $\llbracket S \rrbracket$ is called execution of test case $(I_n, S_{oll})$ if and only if

- there is $n \in \mathbb{N}$ such that $\left( \begin{array}{c} \sigma_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \sigma_n \end{array} \right) \sum_{in} \in \llbracket I_n \rrbracket$.

  (‘A prefix of $\pi$ corresponds to an input sequence’).

Execution $\pi$ of test case $T$ is called

- **successful** (or **positive**) if and only if $\pi \notin \llbracket S_{oll} \rrbracket$.
  - Intuition: an error has been discovered.
  - Alternative: test item $S$ **failed to pass the test**.
  - Confusing: “test failed”.

- **unsuccessful** (or **negative**) if and only if $\pi \in \llbracket S_{oll} \rrbracket$.
  - Intuition: no error has been discovered.
  - Alternative: test item $S$ **passed the test**.
  - Okay: “test passed”.

---
### Test Suite

- **A test suite** is a finite set of test cases \( \{T_1, \ldots, T_n\} \).

- An **execution** of a test suite is a set of computation paths, such that there is at least one execution for each test case.

- An **execution** of a test suite is called **positive** if and only if at least one test case execution is positive. Otherwise, it is called **negative**.

### Not Executing Test Cases

- Consider the test case
  
  \[ T = ("", 0) \]

  for procedure `strlen`.
  ("Empty string has length 0.")

- A tester observes the following software behaviour:
  
  \( \pi = \{ s \mapsto \text{NULL}, r \mapsto 0 \} \leftarrow \text{program-abort} \)

- Test execution **positive** or **negative**?

**Note:**

- If a tester does not adhere to an allowed input sequence of \( T \), \( \pi \) is **not** a test execution. Thus \( \pi \) is neither positive nor negative (only defined for test executions).

- Same case: power outage (if continuous power supply is considered in input sequence).
Tests vs. Systematic Tests

**Test** – (one or multiple) execution(s) of a program on a computer with the goal to find errors. (Ludewig and Lichter, 2013)

**Not (even) a test** (in the sense of this weak definition):
- any inspection of the program (no execution),
- demo of the program (other goal),
- analysis by software-tools for, e.g., values of metrics (other goal),
- investigation of the program with a debugger (other goal).

**Systematic Test** – a test such that
- (environment) conditions are defined or precisely documented,
- inputs have been chosen systematically,
- results are documented and assessed according to criteria that have been fixed before. (Ludewig and Lichter, 2013)

(Our) Synonyms for non-systematic tests: Experiment, 'Rumprobieren'.

In the following: **test** means systematic test; if not systematic, call it **experiment**.

So Simple?
**Environmental Conditions**

*Strictly speaking,* a test case is a triple \((\text{In}, \text{Soll}, \text{Env})\)
comprising a description \(\text{Env}\) of (environmental) conditions.

\(\text{Env}\) describes any aspects which *could have an effect*
on the outcome of a test execution and cannot
be specified as part of \(\text{In}\), such as:

- Which *program* (version) is tested?
- Built with which compiler, linker, etc.?
- Test host (OS, architecture, memory size, connected devices (configuration?), etc.)?
- Which *other software* (in which version, configuration) is involved?
- *Who* is supposed to test *when*?
- etc. etc.

→ test executions should be (as) reproducible and objective (as possible).

**Full reproducibility** is hardly possible *in practice* – obviously (err, why...?).

- **Steps** towards reproducibility and objectivity:
  - have a fixed build environment,
  - use a fixed test host which does not do any other jobs,
  - execute test cases *automatically* (test scripts).

---

**Content (Part II)**

- **Introduction**
  - quotes on testing,
  - systematic testing vs. ‘rumprobieren’.

- **Test Case**
  - definition,
  - execution,
  - positive and negative.

- **Test Suite**

- **Limits of Software Testing**
  - Software examination paths
  - Is exhaustive testing feasible?
  - Range vs. point errors

- **More Vocabulary**
Software Examination (in Particular Testing)

- In each examination, there are two paths from the specification to results:
  - the production path (using model, source code, executable, etc.), and
  - the examination path (using requirements specifications).

- A check can only discover errors on exactly one of the paths.

- If a difference is detected: examination result is positive.

- What is not on the paths, is not checked: crucial: specification and comparison.

Recall:

- Specification
  - Implement specification
  - Comprehend specification
  - Requirements on result

- Comparison
  - Comparison on result

- Information flow development
  - Information flow examination

(Ludewig and Lichter, 2013)
“Software testing can be used to show the presence of bugs,
but never to show their absence!”

(E. W. Dijkstra, 1970)

**Why Can’t We Show The Absence of Errors (in General)?**

Consider a simple pocket calculator for adding 8-digit decimals:

- **Requirement**: If the display shows $x$, $+$, and $y$, then after pressing $=$,
  - the sum of $x$ and $y$ is displayed if $x + y$ has at most 8 digits,
  - otherwise “-E-” is displayed.

- With 8 digits, both $x$ and $y$ range over $[0, 10^8 - 1]$.

- Thus there are $10^{16} = 10,000,000,000,000,000$ possible input pairs $(x, y)$ to be considered for exhaustive testing, i.e. testing every possible case!

- And if we restart the pocket calculator for each test, we do not know anything about problems with sequences of inputs…
  (Local variables may not be re-initialised properly, for example.)
**Observation: Software Usually Has Many Inputs**

- **Example:** Simple Pocket Calculator.
  
  With ten thousand (10,000) different test cases (that's a lot!),
  9,999,999,999,990,000 of the $10^{16}$ possible inputs remain **uncovered**.

  In other words:
  Only $0.0000000001\%$ of the possible inputs are covered, $99.9999999999\%$ not touched.

- **In diagrams:** (red: uncovered, blue: covered)

**Point vs. Range Errors**

- **Software is (in general) not continuous.**
  
  Consider a continuous function, e.g. the one to the right:
  
  For sufficiently small $\epsilon$-environments of an input, the outputs **differ only by a small amount** $\delta$.

- **Physical systems are (to a certain extent) continuous:**
  
  - For example, if a bridge endures a single car of 1000 kg,
    we strongly expect the bridge to endure cars of 990 kg or 1010 kg.
  
  - And anything of weight smaller than 1000 kg can be expected to be endured.

- For software, adjacent inputs **may yield arbitrarily distant** output values.

  **Vocabulary:**
  
  - **Point error:** an isolated input value triggers the error.
  
  - **Range error:** multiple “neighbouring” inputs trigger the error.

- For software, (in general, without extra information) we can not **conclude from some values to others**.
Content (Part II)

• Introduction
  • quotes on testing,
  • systematic testing vs. ‘rumprobieren’.

• Test Case
  • definition,
  • execution.
  • positive and negative.

• Test Suite

• Limits of Software Testing
  • Software examination paths
  • Is exhaustive testing feasible?
  • Range vs. point errors

• More Vocabulary

Tell Them What You’ve Told Them...

• Testing is about
  • finding errors, or
  • demonstrating scenarios.

• A test case consists of
  • input sequences and
  • expected outcome(s).

• A test case execution is
  • positive if an error is found.
  • negative if no error is found.

• A test suite is a set of test cases.

• Distinguish (among others).
  • glass-box test: structure (or source code) of test item available,
  • black-box test: structure not available.
References


