Softwaretechnik / Software-Engineering

Lecture 14: UML State Machines & Software Quality Assurance

2017-07-10

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Architecture & Design: Content

- Introduction and Vocabulary
- Software Modelling I
  (i) views and viewpoints, the 4+1 view
  (ii) model-driven/-based software engineering
  (iii) Modelling structure
    a) (simplified) class diagrams
    b) (simplified) object diagrams
    c) (simplified) object constraint logic (OCL)
    d) Unified Modelling Language (UML)
- Principles of Design
  (i) modularity, separation of concerns
  (ii) information hiding and data encapsulation
  (iii) abstract data types, object orientation
  (iv) Design Patterns
- Software Modelling II
  (i) Modelling behaviour
    a) communicating finite automata
    b) Uppaal query language
    c) basic state-machines
    d) an outlook on hierarchical state-machines
- Testing: Introduction
Recall: CFA, Queries, Model-Checking

Example

Design Verification: Another Invariant

Question: Is it true that, if there is money in the machine and water in stock, then the "water" button is enabled?

Approach: Check 

\[ N_{\text{en}}: \text{have_c50} \lor \text{have_c100} \lor \text{have_c150} \]

implies 

\[ \text{water_enabled} \]

\[ \text{have_c150} \]

\[ \text{have_e1} \]

\[ \text{have_c100} \]

\[ \text{have_c50} \]

\[ \text{idle} \]

OK? OK? OK? OK? OK?

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

\[ N_{\text{en}}: \text{have_c50} \lor \text{have_c100} \lor \text{have_c150} \]

implies 

\[ \text{water_enabled} \]

\[ \text{drink_ready} \]

\[ \text{have_c150} \]

\[ \text{have_e1} \]

\[ \text{have_c100} \]

\[ \text{have_c50} \]

\[ \text{idle} \]

OK? OK? OK? OK? OK?

Example

Satisfaction of UpPAAL Queries by Configurations
**CFA vs. Software**

---

**A CFA Model Is Software**

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

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

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 $[\cdot] : S \rightarrow [S]$ is called interpretation of $S$.

- Let $C(A_1, \ldots, A_n)$ be a network of CFA.
- $\Sigma = Conf$
- $A = Act$
- $[C] = \{ \pi = (\ell_0, v_0) \xrightarrow{\lambda_1} (\ell_1, v_1) \xrightarrow{\lambda_2} (\ell_2, v_2) \xrightarrow{\lambda_3} \cdots | \pi \text{ is a computation path of } C \}$.

**Note:** the structural model just consists of the set of variables and the locations of $C$. 
Content I (Architecture & Design)

- CFA vs. Software
  - a CFA model is software
  - implementing CFA
  - formal methods in the real world: case study

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

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

- Now that we have a CFA model $C(A_1, \ldots, A_n)$ (thoroughly checked using Uppaal), we would like to have executable software – an implementation of the model.

- This task can be split into two sub-tasks:
  (i) implement each CFA $A_i$ in the model by module $S_{A_i}$.
  (ii) implement the communication in the network by module $S_C$.
  (This has, by now, been provided implicitly by the Uppaal simulator and verifier.)

- Fully distributed implementation (without $S_C$): different story, possible for sub-class of CFA.
• Let $N = C(A_1, \ldots, A_n)$ with pairwise disjoint variables.
• Assume $B = B_{\text{input}} \cup B_{\text{internal}}$, where $B_{\text{input}}$ are dedicated input channels, i.e. there is no edge with action $a \in B_{\text{input}}$.
• Then software $S_N$ consists of $S_{A_1}, \ldots, S_{A_n}$ and the following $S_C$.

```c
void main() {
    do □ true :
        (α, snd, rcv) := select(R_1, \ldots, R_n); // choose synchronisation
        // (rcv = 0 if α = τ; blocks on deadlock)
    for (k = 1 to n)
        if □ snd = k :
            R_k := take_action(α) // sender
            □ rcv = k :
                R_k := take_action(¯α) // receiver
        fi
    od
}
```

Example

```c
int w := 3;
typedef {Wi, dispense, W0} st_T;
st_T st := Wi;
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 := W0;
                R := R ∪ \{FILLUP?, DWATER?\};
        fi;
    fi;
    if □ st = dispense :
        if □ α = DOK! ∧ w = 0 :
            st := W0;
            R := R ∪ \{FILLUP!\};
        □ α = DOK! ∧ w > 0 :
            st := W1;
            R := R ∪ \{FILLUP!\};
        fi;
    fi;
    if □ st = W0 :
        if □ α = FILLUP? :
            w := 3;
            st := W1;
            R := R ∪ \{FILLUP?, DWATER?\};
        fi;
    fi;
    return R;
}
```
Translation Scheme...

... for \( \mathcal{A} = (\{\ell_1, \ldots, \ell_m\}, B, \{v_1, \ldots, v_k\}, E, \ell_{\text{ini}}) \) with

\[
E = \{(\ell_1, \alpha_1, \varphi_1, r_1, \ell_1'), \ldots, (\ell_1, \alpha_{1,n_1}, \varphi_{1,n_1}, r_{1,n_1}, \ell_{1,n_1})
\}
\{(\ell_m, \alpha_{m,1}, \varphi_{m,1}, r_{m,1}, \ell_{m,1}), \ldots, (\ell_m, \alpha_{m,n_m}, \varphi_{m,n_m}, r_{m,n_m}, \ell_{m,n_m})\}\:
\]

\[
\begin{align*}
T_1 v_1 & := v_{1,\text{ini}}; \\
t & := \ell_{\text{ini}}; \\
\text{Set}(\text{Act}) \ 	ext{take-action} (\text{Act} \ a) \ {\{ & \\
\text{Set}(\text{Act}) \ R := \emptyset; \ \\
\text{if} \ \\
\ \\
\text{if} \ (\ell_{i,j}' = \ell_{1} \land \varphi_{1}) \ R := R \cup \{\alpha_{1,1}\}; & \\
\text{fi} & \\
\text{fi} & \ \\
\text{fi} & \ \\
\text{fi} & \\
\text{return} \ R; \} & \end{align*}
\]

Deterministic CFA

\textbf{Definition.} A \textbf{network} of CFA \( \mathcal{C} \) with (joint) alphabet \( B \) is called \textbf{deterministic} if and only if each reachable configuration has at most one successor configuration, i.e. if

\[
\forall c \in \text{Conf}(\mathcal{C}) \text{ reachable} \forall \lambda \in B \cup \{\tau\} \forall c_1, c_2 \in \text{Conf}(\mathcal{C}) \bullet
\]

\[
c \xrightarrow{\lambda} c_1 \land c \xrightarrow{\lambda} c_2 \implies c_1 = c_2.
\]

\textbf{Proposition.} Whether \( \mathcal{C} \) is deterministic is \textbf{decidable}.

\textbf{Proposition.} If \( \mathcal{C} \) is deterministic, then the translation of \( \mathcal{C} \) is a \textbf{deterministic program}.
• Define $[S_N]$ to be the set of computation paths $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$
such that $\sigma_i$ has the values at 'snapshot' at the $i$-th iteration and $\alpha_i$ is the $i$-th action.

• Then $[S_N]$ bisimulates the behaviour $[C]$ of model $C(A_1, \ldots, A_n)$.

Yes, and...?

• If Uppaal reports that $\mathcal{N}_{\forall} \models \exists w = 0$ holds, then $w = 0$ (should be) reachable in $[S_{\forall}]$.

• If Uppaal reports that $\mathcal{N}_{\forall} \models \forall \Box \text{tea\_enabled} \implies \text{CoinValidator\_have\_c150}$ holds, then $[S_{\forall}]$ (should be) correspondingly safe.

In General: If Uppaal reports that

• a desired configuration is not reachable in the model, or

• an invariant does not hold in the model,

then there is an issue with the model, or the requirement (or the checking tool) to be investigated.
• (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.

Content I (Architecture & Design)

• CFA vs. Software
  • a CFA model is software
  • implementing CFA
  • formal methods in the real world: case study

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

• Unified Modelling Language
  • Brief History
  • Sub-Languages
  • UML Modes
Case Study: Wireless Fire Alarm System

(R1) The **loss of the ability** of the system to transmit a signal from a component to the central unit is **detected** in less than 300 seconds [...].

\[ \wedge_{i \in C} \Box (FAIL = i \land \neg DET_i) \implies \ell \leq 300s \]

(R2) A **single alarm event** is **displayed** at the central unit within 10 seconds.

\[ \wedge_{i \in C} \lceil ALARM \rceil_i \implies \Box (ALARM_i \land \neg DISP_i) \implies \ell \leq 10s , \]
### Process Model

![Diagram of the process model](image)

#### Figures (Arenis et al., 2016)

<table>
<thead>
<tr>
<th>Monitoring</th>
<th>Templates</th>
<th>Instances</th>
<th>Total Locations</th>
<th>Clocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensors as slaves</td>
<td>9</td>
<td>15</td>
<td>1040</td>
<td>6</td>
</tr>
<tr>
<td>Repeaters as slaves</td>
<td>9</td>
<td>21</td>
<td>82</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>Sensors as slaves, $N = 126$</th>
<th>Repeaters as slaves, $N = 10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 Detection possible</td>
<td>10.205.13</td>
<td>357.00</td>
</tr>
<tr>
<td>Q2 No message collision</td>
<td>12.895.17</td>
<td>2343.00</td>
</tr>
<tr>
<td>Q3 All set downlock</td>
<td>36.070.78</td>
<td>3149.00</td>
</tr>
<tr>
<td>Q4 Nop</td>
<td>97.14</td>
<td>44.25</td>
</tr>
</tbody>
</table>

**Verification of the final design (Opteron 6174 2.2Ghz, 64GB, Uppaal 4.1.3 (64-bit), options -s -t0 -u).**

<table>
<thead>
<tr>
<th>Model</th>
<th>sequential</th>
<th>optimized</th>
<th>test scenario</th>
<th>Measured Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Alarm</td>
<td>3.20s</td>
<td>2.14s</td>
<td>3.31s</td>
<td>2.79s ± 0.53s</td>
</tr>
<tr>
<td>All 10 Alarms</td>
<td>29.03s</td>
<td>27.08s</td>
<td>29.81s</td>
<td>29.65s ± 3.26s</td>
</tr>
</tbody>
</table>

Predicted alarm transmission times vs. Measurements on real hardware.

**Content I (Architecture & Design)**

- **CFA vs. Software**
  - a CFA model is software
  - implementing CFA
  - formal methods in the real world: case study

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

- **Unified Modelling Language**
  - Brief History
  - Sub-Languages
  - UML Modes
UML Core State Machines

\[ s_1 \quad E/itsD!F \quad s_2 \]

\[ s_3 \quad /x := 0 \quad /itsC!G \]

\[ annot := \left[ \langle event\rangle, \langle event\rangle \right]^{*} \quad \left[ \langle guard\rangle \right] \quad \left[ I\langle action\rangle \right] \]

with

- event \in E,
- guard \in Expr, (default: true, assumed to be in Expr)
- action \in Act, (default: skip, assumed to be in Act)
### Event Pool and Run-To-Completion

<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>$s_1$</td>
<td>1</td>
<td>$E$ ready for $u_1$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$s_2$</td>
<td>1</td>
<td>$s_1$</td>
<td>1</td>
<td>$F$ ready for $u_2$</td>
<td></td>
</tr>
</tbody>
</table>
### Event Pool and Run-To-Completion

<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>27</td>
<td>( s_1 )</td>
<td>1</td>
<td>( E ) ready for ( u_1 )</td>
</tr>
<tr>
<td>1</td>
<td>( s_2 )</td>
<td>1</td>
<td>27</td>
<td>( s_1 )</td>
<td>1</td>
<td>( F ) ready for ( u_2 )</td>
</tr>
<tr>
<td>2</td>
<td>( s_2 )</td>
<td>1</td>
<td>27</td>
<td>( s_2 )</td>
<td>0</td>
<td>( G ) ready for ( u_3 )</td>
</tr>
</tbody>
</table>

The table and diagram illustrate the transitions and conditions in the event pool for the given states and variables.

---

*Note: The diagram represents the transitions and states, with arrows indicating the flow and conditions for each step.*
Event Pool and Run-To-Completion

<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>s₁</td>
<td>1</td>
<td>27</td>
<td>s₁</td>
<td>1</td>
<td>E ready for u₁</td>
</tr>
<tr>
<td>1</td>
<td>s₂</td>
<td>1</td>
<td>27</td>
<td>s₁</td>
<td>1</td>
<td>F ready for u₂</td>
</tr>
<tr>
<td>2</td>
<td>s₂</td>
<td>1</td>
<td>27</td>
<td>s₂</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s₃</td>
<td>1</td>
<td>27</td>
<td>s₃</td>
<td>0</td>
<td>G ready for u₃</td>
</tr>
<tr>
<td>4.a</td>
<td>s₂</td>
<td>1</td>
<td>0</td>
<td>s₁</td>
<td>1</td>
<td>G ready for u₁</td>
</tr>
</tbody>
</table>

Event Pool and Run-To-Completion

<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>s₁</td>
<td>1</td>
<td>27</td>
<td>s₁</td>
<td>1</td>
<td>E ready for u₁</td>
</tr>
<tr>
<td>1</td>
<td>s₂</td>
<td>1</td>
<td>27</td>
<td>s₁</td>
<td>1</td>
<td>F ready for u₂</td>
</tr>
<tr>
<td>2</td>
<td>s₂</td>
<td>1</td>
<td>27</td>
<td>s₂</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s₃</td>
<td>1</td>
<td>27</td>
<td>s₃</td>
<td>0</td>
<td>G ready for u₃</td>
</tr>
<tr>
<td>4.a</td>
<td>s₂</td>
<td>1</td>
<td>0</td>
<td>s₁</td>
<td>1</td>
<td>G ready for u₁</td>
</tr>
<tr>
<td>5.a</td>
<td>s₁</td>
<td>1</td>
<td>0</td>
<td>s₁</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
### Event Pool and Run-To-Completion

<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>$27$s_1</td>
<td>1</td>
<td>E ready for $u_1$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$s_2$</td>
<td>1</td>
<td>$27$s_1</td>
<td>1</td>
<td>F ready for $u_2$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$s_2$</td>
<td>1</td>
<td>$27$s_2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$s_2$</td>
<td>1</td>
<td>$27$s_3</td>
<td>0</td>
<td>G ready for $u_1$</td>
<td></td>
</tr>
<tr>
<td>4.a</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>5.a</td>
<td>$s_1$</td>
<td>1</td>
<td>0</td>
<td>$s_1$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4.b</td>
<td>$s_1$</td>
<td>1</td>
<td>$27$s_3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.b</td>
<td>$s_1$</td>
<td>1</td>
<td>0</td>
<td>$s_1$</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
"D just stepped from $s_1$ to $s_2$ by transition $r$"
Composite (or Hierarchical) States

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

Example
→ "Software Design, Modelling, and Analysis with UML" in some winter semesters.

Content I (Architecture & Design)

- **CFA vs. Software**
  - a CFA model is software
  - implementing CFA
  - formal methods in the real world: case study

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

- **Unified Modelling Language**
  - Brief History
  - Sub-Languages
  - UML Modes
• Boxes/lines and finite automata are used to visualise software for ages.

• **1970’s, Software Crisis™**
  – Idea: learn from engineering disciplines 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)
Boxes/lines and finite automata are used to visualise software for ages.

1970’s, Software Crisis™
– Idea: learn from engineering disciplines 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…
• Boxes/lines and finite automata are used to visualise software for ages.

• 1970's, Software Crisis™
  – Idea: learn from engineering disciplines 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.

---

**UML Overview** *(OMG, 2007b, 684)*

![Diagram](image)

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

*Dobing and Parsons (2006)*
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](image)

- **Floorplan as blueprint:**
  ![Floorplan as blueprint](image)

- **Floorplan as program:**
  ![Floorplan as program](image)
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.

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. [...]

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.
• The “mode” fitting the lecture best is **AsBlueprint**.

**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 the standard documents **OMG (2007a,b)** as far as possible.

**Plus:**

• Being precise also helps to work in mode **AsSketch**:
  Knowing “the real thing” should make it easier to
  (i) “see” which blueprint(s) the sketch is supposed to denote, and
  (ii) to ask meaningful questions to resolve ambiguities.

---

**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.)

• Mind **UML Modes.**
Code Quality Assurance

Topic Area Code Quality Assurance: Content

- Introduction and Vocabulary
  - Test case, test suite, test execution.
  - Positive and negative outcomes.

- Limits of Software Testing
  - Glass-Box Testing
    - Statement-, branch-, term-coverage.

- Other Approaches
  - Model-based testing,
  - Runtime verification.

- Software quality assurance in a larger scope.

- Program Verification
  - Partial and total correctness.
  - Proof System PD.

- Review
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 $[[S]]$ of (finite or infinite) computation paths of the form $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \ldots$ 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 $[[ \cdot ]] : S \mapsto [[S]]$ 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
  $$\pi = \left( \begin{array}{c} \sigma_0 \\ \sigma_0' \end{array} \right) \xrightarrow{\alpha_1} \left( \begin{array}{c} \sigma_1 \\ \sigma_1' \end{array} \right) \xrightarrow{\alpha_2} \ldots$$

- We use $\pi \downarrow \Sigma_{\text{in}}$ to denote $\pi = \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \ldots$, i.e. the projection of $\pi$ onto $\Sigma_{\text{in}}$. 
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 $[\cdot]$ of these descriptions:

- $[In] \subseteq (\Sigma_{in} \times A)^*$, $[Soll] \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$

Examples:

- Test case for procedure strlen : $String \to N$, $s$ denotes parameter, $r$ return value:

  $T = (s = "abc", r = 3)$

  $[s = "abc"] = \{ \sigma^0 \xrightarrow{s} \cdot \cdots \xrightarrow{\tau} \sigma^1 | \sigma_0(s) = "abc" \}$

  $[r = 3] = \{ \sigma^0 \xrightarrow{\tau} \sigma^1 | \sigma_1(r) = 3 \}$

  Shorthand notation: $T = ("abc", 3)$.

- “Call 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 $[\cdot]$ of these descriptions:

- $[In] \subseteq (\Sigma_{in} \times A)^*$, $[Soll] \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$

Examples:

- Test case for vending machine.

  $T = (C50, WATER; DWATER)$

- “Send event $C50$ and any time later $WATER$, expect $DWATER$ after 10 steps the latest.”
**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:
- $\llbracket In \rrbracket \subseteq (\Sigma_{in} \times A)^*$,
- $\llbracket Soll \rrbracket \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$

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
  \[
  \pi = (\sigma_0^0, \sigma_0^1) \xrightarrow{\alpha_1} (\sigma_1^0, \sigma_1^1) \xrightarrow{\alpha_2} \ldots
  \]
  from $\llbracket S \rrbracket$ is called execution of test case $(In, Soll)$ if and only if
  - there is $n \in \mathbb{N}$ such that $\sigma_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \sigma_n \downarrow \Sigma_{in} \in \llbracket In \rrbracket$.
  (A prefix of $\pi$ corresponds to an input sequence).

Execution $\pi$ of test case $T$ is called
- **successful** (or **positive**) if and only if $\pi \notin \llbracket Soll \rrbracket$.
  - Intuition: an an error has been discovered.
  - Alternative: test item $S$ failed to pass the test.
  - Confusing: “test failed”.

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


• Consider the test case  

\[ T = (\"\", 0) \]

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

• A tester observes the following software behaviour: 

\[ \pi = \{ s \mapsto \text{NULL}, r \mapsto 0 \} \]

\[ \xrightarrow{\delta_0} \]  

\[ T \rightarrow \text{program-abortion} \]

\[ \xrightarrow{\delta_1} \]

• Test execution positive or negative?

---

**By The Way... (Good Design)**

• **High quality software** should be aware of its specification.  
  and "complain" if operated outside of specification, e.g.  
  • throw an exception.  
  • abort program execution.  
  • (at least) print an error message.  
  • etc.

  **Not:** "garbage in, garbage out"

• **Example:** strlen(3) (C standard)  
  • Allowed inputs are C-strings, return value is an integer,  
  • NULL is not a C-string!  
  • Thus, on input NULL, "complain" instead of just return an arbitrary number.
**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**.

---

**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


