Introduction and Vocabulary

Principles of Design

(i) modularity
(ii) separation of concerns
(iii) information hiding and data encapsulation
(iv) abstract data types, object orientation

Software Modelling

(i) views and viewpoints, the 4+1 view
(ii) model-driven/-based software engineering
(iii) Unified Modelling Language (UML)
(iv) modelling structure
   a) (simplified) class diagrams
   b) (simplified) object diagrams
   c) (simplified) object constraint logic (OCL)
(v) modelling behaviour
   a) communicating finite automata
   b) Uppaal query language
   c) implementing CFA
   d) an outlook on UML State Machines

Design Patterns

Testing: Introduction
## Content

- **CFA at Work** continued
  - design checks and verification
  - Uppaal architecture
  - case study

- **CFA vs. Software**
  - a CFA model is software
  - implementing CFA
  - Recall MDSE

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

- **UML Modes**
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
  \[ \mathcal{N}_{VM} \models \exists \diamond w = 0. \]
  for the vending machine model \( \mathcal{N}_{VM} \).
**Question:** Is the following existential LSC satisfied by the model? (Otherwise, the design is definitely broken.)

**Approach:** Use the following newly created CFA ‘Scenario’ instead of User and check whether location end_of_scenario is reachable, i.e. check

\[ \mathcal{N}_{VM}' \models \exists \Diamond \text{Scenario.end_of_scenario}. \]

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

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

holds in all reachable configurations, i.e. check

\[
N_{VM} \models \forall \Box \text{tea\_enabled} \implies \text{CoinValidator\_have\_c150}
\]

for the vending machine model $N_{VM}$. 

[Diagram of a vending machine model]
Design Verification: Sanity Check

- **Question**: Is the “tea” button ever enabled?
  (Otherwise, the considered invariant
  \[ \text{tea} \_\text{enabled} \implies \text{CoinValidator} . \text{have} \_\text{c150} \]
  holds vacuously.)

- **Approach**: Check whether a configuration satisfying \( \text{water} \_\text{enabled} = 1 \) 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

\[ \mathcal{N}_{VM} \models \forall \Box \left( \text{CoinValidator.have}_c_{50} \text{ or CoinValidator.have}_c_{100} \text{ or CoinValidator.have}_c_{150} \right) \]

imply water\_enabled.
Recall: Universal LSC Example

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

Diagram:

- User
- CoinValidator
- ChoicePanel
- Dispenser

Conditions:
- water in stock
- ¬(C50 ∨ E1 ∨ pSOFT! ∨ pTEA! ∨ pFILLUP!)
- ¬(dSoft ∨ dTEA!)

Actions:
- c50
- pWATER
- dWATER
- OK
Uppaal Architecture
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 [...].

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

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

\[ \wedge_{i \in C} \square (\overline{ALARM_i} \land \neg DISP_i) \implies \ell \leq 10s, \]
Content

- **CFA at Work** continued
  - design checks and verification
  - Uppaal architecture
  - case study

- **CFA vs. Software**
  - a CFA model is software
  - implementing CFA
  - Recall MDSE

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

- **UML Modes**
CFA vs. Software
A CFA Model Is 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.
- $\Sigma = \text{Conf}$
- $A = \text{Act}$
- $\llbracket C \rrbracket = \{ \pi = (\vec{\ell}_0, \nu_0) \xrightarrow{\lambda_1} (\vec{\ell}_1, \nu_1) \xrightarrow{\lambda_2} (\vec{\ell}_2, \nu_2) \xrightarrow{\lambda_3} \cdots \mid \pi \text{ is a computation path of } C \}$.

- **Note**: the structural model just consists of the set of variables and the locations of $C$. 
**Formal Methods in the Software Development Process**

Customer 2

\[ \mathcal{S}_1 = \{(M.C, [\cdot]_1), (C.M, [\cdot]_1)\} \]

\[ \mathcal{S}_2 = \{(M.T.M.C, [\cdot]_1), (C.T.C.M, [\cdot]_1)\} \]

\[ S = \{\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \ldots \} \]
Model-Driven Software Engineering

- (Jacobson et al., 1992): “System development is model building.”
- Model **driven** software engineering (MDSE): **everything** is a model.
- Model **based** software engineering (MBSE): **some** models are used.
Implementing CFA

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

- This task can be split into two sub-tasks:
  1. **implement** each CFA $A_i$ in the model by module $S_{A_i}$.
  2. **implement** the communication in the network by module $S_C$.

(This has, by now, been provided implicitly by the Uppaal simulator and verifier.)
Example

\[ w := 3 \]

\[ w > 0 \quad \text{DOK!} \]

\[ w == 0 \quad \text{DOK!} \]

\[ w := w - 1 \]

\[ \text{FILLUP?} \]

\[ \text{DOK!} \]

\[ \text{dispense} \]

\[ \text{W0} \]

\[ \text{st : \{} \text{Wi, W0, dispense} \}\]

\[ w : \text{int} = 3 \]

\[ \text{\square st = Wi + if (\alpha = \text{DWRITE})} \]

\[ \text{w := w - 1; st := dispense; else (\alpha = \text{FILLUP})} \]

\[ \text{w := 3;} \]
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 := Wi;
      R := R ∪ {FILLUP?, DWATER?};
  fi;

  □ st = dispense :
    if □ α = DOK! ∧ w = 0 :
      □ α = DOK! ∧ w > 0 :
        □ α = FILLUP? :
          w := 3;
st := Wi;
          R := R ∪ {FILLUP?, DWATER?};
      fi;

      □ α = DOK! ∧ w > 0 :
        □ α = FILLUP? :
          w := 3;
st := Wi;
          R := R ∪ {FILLUP?, DWATER?};
      fi;

      □ α = DOK! ∧ w < 0 :
        □ α = FILLUP? :
          w := 3;
st := Wi;
          R := R ∪ {FILLUP?, DWATER?};
      fi;

    fi;
  return R;
}
```
... for $A = (\{\ell_1, \ldots, \ell_m\}, B, \{v_1, \ldots, v_k\}, E, \ell_{ini})$ with

$$E = \{((\ell_1, \alpha_{1,1}, \varphi_{1,1}, \bar{r}_{1,1}, \ell'_{1,1}), \ldots, (\ell_1, \alpha_{1,n_1}, \varphi_{1,n_1}, \bar{r}_{1,n_1}, \ell'_{1,n_1}),$$

$$\ldots,$$

$$(\ell_m, \alpha_{m,1}, \varphi_{m,1}, \bar{r}_{m,1}, \ell'_{m,1}), \ldots, (\ell_m, \alpha_{m,n_m}, \varphi_{m,n_m}, \bar{r}_{m,n_m}, \ell'_{m,n_m})\} :$

$$T_1 v_1 := v_{1,ini}; \ldots T_k v_k := v_{k,ini};$$

```c
typedef \{\ell_1, \ldots, \ell_m\} st_T;
st_T \text{st} := \ell_{ini};
Set\langle Act\rangle \text{take\_action}(\text{Act} \alpha) \{ 
Set\langle Act\rangle R := \emptyset;
if
\ldots,
\Box \text{st} = \ell_i : \ \text{if}
\ldots,
\Box \alpha = \alpha_{i,j} \land \varphi_{i,j} : \bar{r}_{i,j};
\text{st} := \ell'_{i,j};
\text{if} (\ell'_{i,j} = \ell_1 \land \varphi_{1,1}) R := R \cup \{\alpha_{1,1}\};
\ldots
\text{if} (\ell'_{i,j} = \ell_m \land \varphi_{m,n_m}) R := R \cup \{\alpha_{m,n_m}\};
\ldots
fi;
\ldots
\} fi;
return R;
```
**Deterministic CFA**

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

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

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

**Proposition.** Whether $C$ is deterministic is decidable.

**Proposition.** If $C$ is deterministic, then the translation of $C$ is a deterministic program.
Let $\mathcal{N} = \mathcal{C}(A_1, \ldots, A_n)$ 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 $a!$ and $a \in B_{\text{input}}$.

Then software $S_\mathcal{N}$ consists of $S_{A_1}, \ldots, S_{A_n}$ and the following $S_C$.

```plaintext
Set\langle\text{Act}\rangle R_1 := R_{1,\text{ini}}, \ldots, R_n := R_{n,\text{ini}}; \hspace{1em} // \text{initially enabled actions}

void main() {
    do
        □ true : if
            □ true : (α, snd, rcv) := select(R_1, \ldots, R_n); \hspace{1em} // \text{choose synchronisation}
            \hspace{1em} // \hspace{1em} rcv = 0 if α = τ,
            \hspace{1em} // \hspace{1em} blocks on deadlock
        □ true : (α, snd, rcv) := read_input(); \hspace{1em} // or read input (snd = 0)
        fi
    for (k = 1 to n) if (snd = k) R_k := take_{\text{action}}_k(α); \hspace{1em} // sender
    for (k = 1 to n) if (rcv = k) R_k := take_{\text{action}}_k(\bar{α}); \hspace{1em} // receiver
    od \hspace{1em} // \text{snapshot}
}
```
Model vs. Implementation

- 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 $\mathcal{T}(C(A_0, A_1, \ldots, A_n))$ where $A_0$ has one location $\ell$ and edges $E_0 = \{(\ell, \alpha!, \text{true}, \langle\rangle, \ell) \mid \alpha \in B_{\text{input}}\}$.

\[ \begin{aligned} B_{\text{input}} & = \{\text{C50, WATER, \ FILLUP}\} \\ A_0: & \begin{array}{c} \text{idle} \\ \text{C50!} \quad \text{WATER!} \quad \text{FILLUP!} \end{array} \\ A_1: & \begin{array}{c} \text{idle} \\ \text{C50?} \quad \text{have_c50} \quad \text{WATER?} \quad \text{water_selected} \quad \text{DOK?} \quad \text{D WATER!} \end{array} \\ A_2: & \begin{array}{c} \text{Wi} \\ \text{w > 0} \quad \text{DOK!} \quad \text{DWATER?} \quad \text{w := w - 1} \quad \text{FILLUP?} \quad \text{w := 3} \end{array} \end{aligned} \]

\[ \begin{aligned} J(C): & \begin{array}{c} \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \\ \text{w := w} \end{array} \\ L_S(C): & \begin{array}{c} \tilde{\sigma}_0 \xrightarrow{\tilde{\alpha}_1} \tilde{\sigma}_1 \xrightarrow{\tilde{\alpha}_2} \tilde{\sigma}_2 \cdots \\ \text{w := w} \end{array} \end{aligned} \]
Model vs. Implementation

- 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 $\mathcal{T}(\mathcal{C}(A_0, A_1, \ldots, A_n))$ where $A_0$ has one location $\ell$ and edges $E_0 = \{(\ell, \alpha!, true, \langle\rangle, \ell) \mid \alpha \in B_{input}\}$.

Yes, and...?

- If Uppaal reports that $\mathcal{N}_{VM} \models \exists \diamond w = 0$ holds, then $w = 0$ is reachable in $[S_{\mathcal{N}_{VM}}]$.

- If Uppaal reports that $\mathcal{N}_{VM} \models \forall \square tea\_enabled \implies \text{CoinValidator.have\_c150}$ holds, then $[S_{\mathcal{N}_{VM}}]$ is correspondingly safe.
Model-Driven Software Engineering

- (Jacobson et al., 1992): “System development is model building.”
- Model **driven** software engineering (MDSE): **everything** is a model.
- Model **based** software engineering (MBSE): **some** models are used.
Content

• CFA at Work continued
  • design checks and verification
  • Uppaal architecture
  • case study

• CFA vs. Software
  • a CFA model is software
  • implementing CFA
  • Recall MDSE

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

• UML Modes
UML State Machines
**UML Core State Machines**

\[ \text{annot} ::= [\langle \text{event} \rangle . \langle \text{event} \rangle]^* \quad [\langle \text{guard} \rangle] \quad [\langle \text{action} \rangle] \]

with
- \( \text{event} \in \mathcal{E} \),
- \( \text{guard} \in \text{Expr} \mathcal{F} \) (default: \( \text{true} \), assumed to be in \( \text{Expr} \mathcal{F} \))
- \( \text{action} \in \text{Act} \mathcal{F} \) (default: \( \text{skip} \), assumed to be in \( \text{Act} \mathcal{F} \))
Event Pool and Run-To-Completion

\( u_1 : C \quad \rightarrow \quad u_2 : D \)

\[ \begin{align*}
\text{stable} & \quad \text{state} \\
\text{it'sD} & \quad \text{it'sC} \\
\text{st} & \quad \text{st} \\
1 & \quad 27 \\
\text{e/itsD} & \quad /x := 0 \\
\rightarrow G & \quad \rightarrow /\text{itsC!G} \\
\end{align*} \]

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

\[ E/\text{itsD!F} \]

\[ F \text{ ready for } u_2 \]

\[ F[x > 0] \]

\[ /x := 0 \]

\[ /\text{itsC!G} \]

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

**Diagram:**

- **Node 1:** $s_1$
- **Node 2:** $s_2$
- **Edge 1:** $E/\text{itsD}!F$
- **Edge 2:** $F[x > 0]$
- **Node 3:** $s_3$
- **Edge 3:** $/\text{itsC}!G$

**Table:**

<table>
<thead>
<tr>
<th>Step</th>
<th>$u_1$ state</th>
<th>$u_1$ stable</th>
<th>$x$</th>
<th>$u_2$ state</th>
<th>$u_2$ 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></td>
</tr>
<tr>
<td>3</td>
<td>$s_2$</td>
<td>1</td>
<td>27</td>
<td>$s_3$</td>
<td>0</td>
<td>$G$ ready for $u_1$</td>
</tr>
</tbody>
</table>
### Event Pool and Run-To-Completion

#### Diagram

- **Event Pool**
  - Event E ready for u₁
  - Event F ready for u₂
  - Event G ready for u₁

#### 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>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>s₂</td>
<td>1</td>
<td>27</td>
<td>s₃</td>
<td>0</td>
<td>G ready for u₁</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>
<tr>
<td>4.b</td>
<td>s₁</td>
<td>1</td>
<td>27</td>
<td>s₃</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5.b</td>
<td>s₁</td>
<td>1</td>
<td>0</td>
<td>s₁</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- **Disjoint Event Pool**
  - Event H ready for u₃

- **Environment**
  - F[x > 0]
  - /x := 0

- **State Diagram**
  - States: s₁, s₂
  - Transitions:
    - s₁ → G
    - s₂ → E
    - s₁ → F
    - /s₁ → /itsD ! F
    - /s₂ → /itsC ! G

- **Constraints**
  - x := 0
  - x > 0

- **Arrows**
  - Red arrows indicate transitions between states.
  - Blue arrows indicate event readiness.

- **Symbols**
  - E: Event
  - F: Event
  - G: Event
  - /: Disjoint Event
  - !: Event Readiness
Rhapsody Architecture

generate

build / make
(compiler)

run

C.h
C.cpp
MainDefaultComponent.cpp

D.h
D.cpp

1

1

generate

build / make
(compiler)

run

DfltCmp.exe
“D just stepped from $s_1$ to $s_2$ by transition $t$”

Rhapsody Architecture

generate

C.h  D.h  C.cpp  D.cpp  MainDefaultComponent.cpp

build / make

(.compiler)

run

DfltCmp.exe
Composite (or Hierarchical) States

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

Entry Action:
itsChoicePanel
->enable_Water();

have_c100_or_e1>

Entry Action:
itsChoicePanel
->enable_Soft();

Entry Action:
itsChoicePanel
->enable_Tea();

Idle

C50

drinkReady

waitOK

have_c50>

have_c100

have_c150>

E1/itsChanger
->giveback_100()

E1/itsChanger
->giveback_100()

C50/itsChoicePanel
->enable_Water();

C50/itsChoicePanel
->enable_Water();

E1/itsChanger
->giveback_100()

C50

C50/itsChanger
->giveback_50()

C50

E1

Entry Action:
itsChoicePanel
->enable_Soft();

Entry Action:
itsChoicePanel
->enable_Tea();

drinkReady

C50

OK

WATER[Water_enabled]

W1

DATER/Prepare_Water();
itsCoinValidator
->GEN(OK);

W2

DATER/Prepare_Water();
itsCoinValidator
->GEN(OK);

W1

DATER/Prepare_Water();
itsCoinValidator
->GEN(OK);

WATER/Prepare_Water();
itsCoinValidator
->GEN(OK);

SOFT[Soft_enabled]

S1

Dнтер/Prepare_Soft();
itsCoinValidator
->GEN(OK);

S2

Dнтер/Prepare_Soft();
itsCoinValidator
->GEN(OK);

S1

Dнтер/Prepare_Soft();
itsCoinValidator
->GEN(OK);

SOFT/Prepare_Soft();
itsCoinValidator
->GEN(OK);

Tea_selected

T2

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T3

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T1

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

Tea_selected

inactive

W3

DWATER/Prepare_Water();
itsCoinValidator
->GEN(OK);

S3

DWATER/Prepare_Water();
itsCoinValidator
->GEN(OK);

S1

DWATER/Prepare_Water();
itsCoinValidator
->GEN(OK);

FILLUP/itsCoinValidator
->update_ChoicePanel();

on

on

on

T2

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T3

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T1

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T2

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T3

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

on

T2

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T3

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T1

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T2

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T3

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

on

on

T2

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T3

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T1

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T2

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T3

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

on

on

T2

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T3

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

T1

Dнтер/Prepare_Tea();
itsCoinValidator
->GEN(OK);

FILLUP/itsCoinValidator
->update_ChoicePanel();

ok

ok

ok
→ “Software Design, Modelling, and Analysis with UML” in the winter semester.
UML Modes
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 blueprint:**
- **Floorplan as program:**
  + wiringplan
  + windows
  + ...
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.
The last slide is inspired by Martin Fowler, who puts it like this:

**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. [...] 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. [...]
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.
References


