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

VL 10	• **Introduction and Vocabulary**
	• **Software Modelling I**
⋮	(i) views and viewpoints, the 4+1 view
	(ii) model-driven/-based software engineering
VL 11	(iii) **Modelling structure**
	a) (simplified) class diagrams
⋮	b) (simplified) object diagrams
VL 12	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**
VL 13	• **Software Modelling II**
	(i) **Modelling behaviour**
⋮	a) communicating finite automata
	b) Uppaal query language
VL 14	c) basic state-machines
⋮	d) an outlook on hierarchical state-machines
	• **Testing**: Introduction

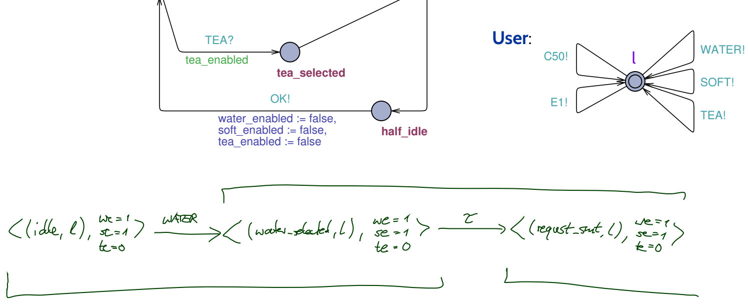
Content I (Architecture & Design)

- **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

Recall: CFA, Queries, Model-Checking

Example

ChoicePanel: (simplified)



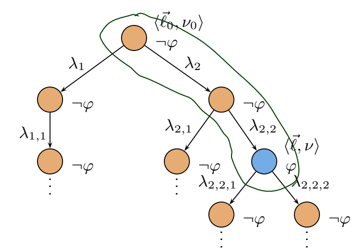
Satisfaction of Uppaal Queries by Configurations

Exists finally:

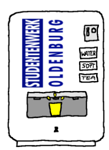
- $\langle \vec{\ell}_0, \nu_0 \rangle \models \exists\Diamond\, term$ iff $\exists$ path ξ of $\mathcal{N}$ starting in $\langle \vec{\ell}_0, \nu_0 \rangle$ $\exists i \in \mathbb{N}_0 \bullet \xi^i \models term$ — *i-th configuration of ξ*

"some configuration satisfying $term$ is reachable"

Example: $\langle \vec{\ell}_0, \nu_0 \rangle \models \exists\Diamond\, \varphi$

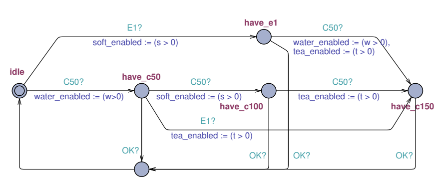


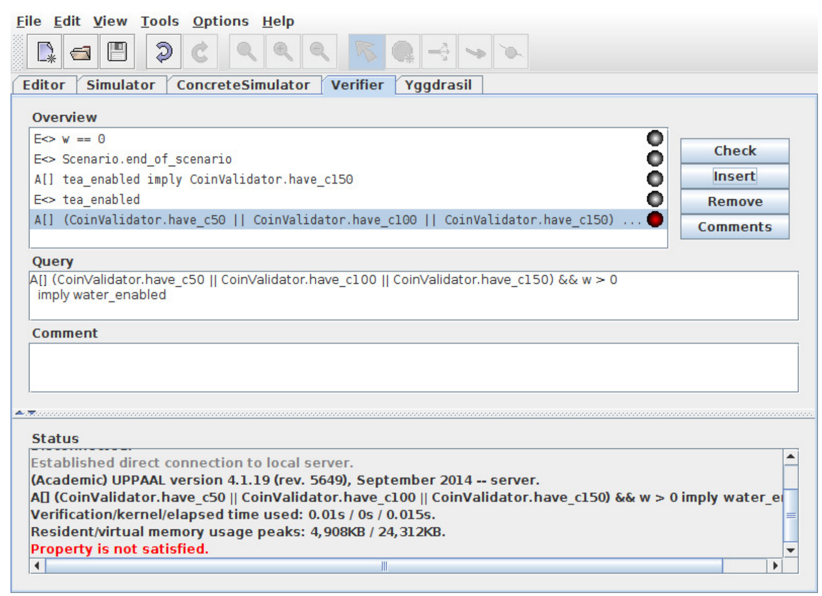
Design Verification: Another Invariant



- **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

$\mathcal{N}_{\mathrm{VM}} \models \forall\Box$ (CoinValidator.`have_c50` or CoinValidator.`have_c100` or CoinValidator.`have_c150`) *and w > 0*
`imply water_enabled`.





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

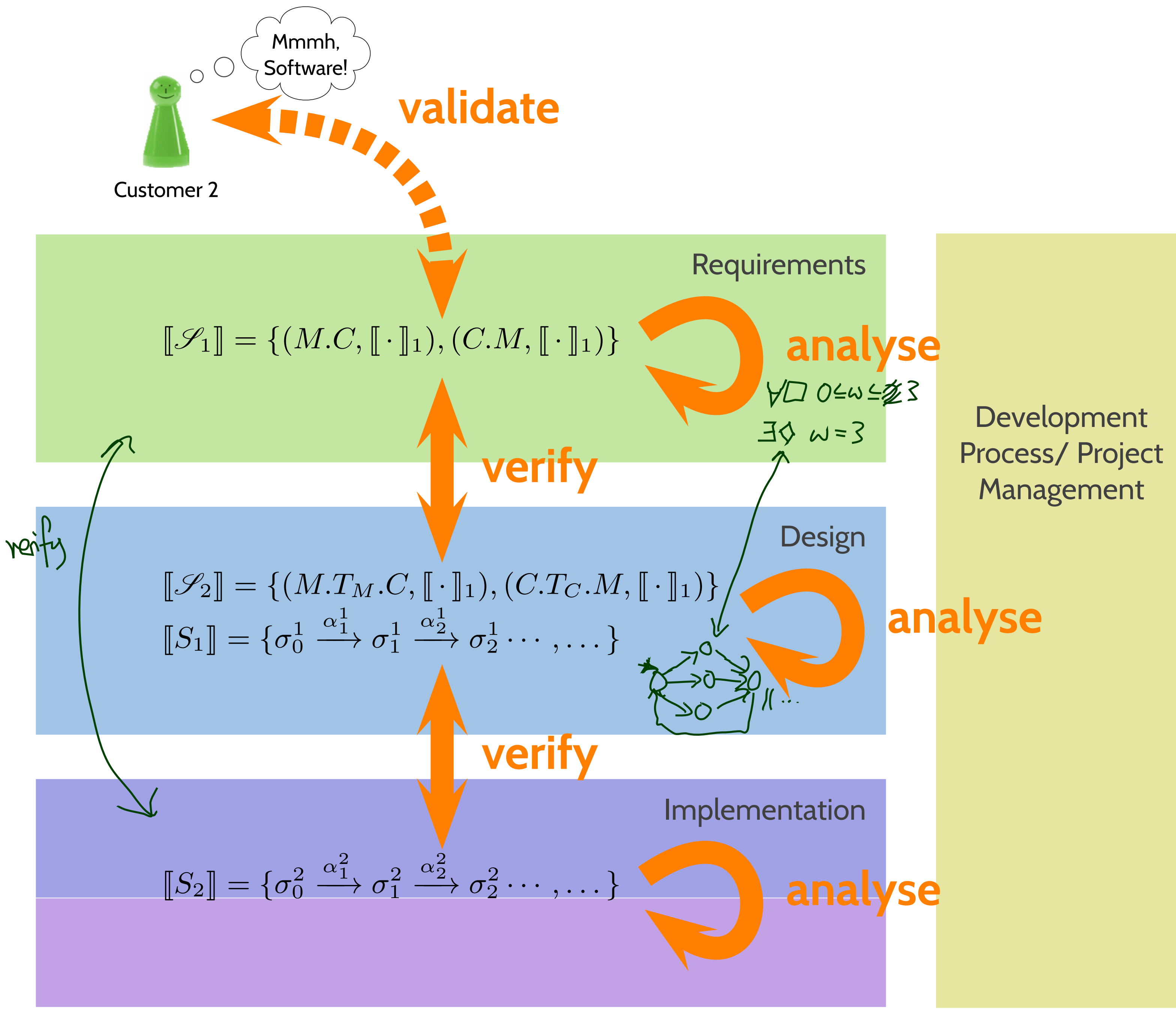
- $\sigma_i \in \Sigma$, $i \in \mathbb{N}_0$, is called state (or configuration), and
- $\alpha_i \in A$, $i \in \mathbb{N}_0$, is called action (or event).

The (possibly partial) function $[\![\cdot]\!] : S \mapsto [\![S]\!]$ is called interpretation of S.

- Let $\mathcal{C}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ be a network of CFA.
- $\Sigma = Conf$
- $A = Act$
- $[\![\mathcal{C}]\!] = \{\pi = \langle \vec{\ell}_0, \nu_0 \rangle \xrightarrow{\lambda_1} \langle \vec{\ell}_1, \nu_1 \rangle \xrightarrow{\lambda_2} \langle \vec{\ell}_2, \nu_2 \rangle \xrightarrow{\lambda_3} \cdots \mid \pi$ is a computation path of $\mathcal{C}\}$.

- **Note**: the structural model just consists of the set of variables and the locations of $\mathcal{C}$.

Formal Methods in the Software Development Process



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

Implementing CFA

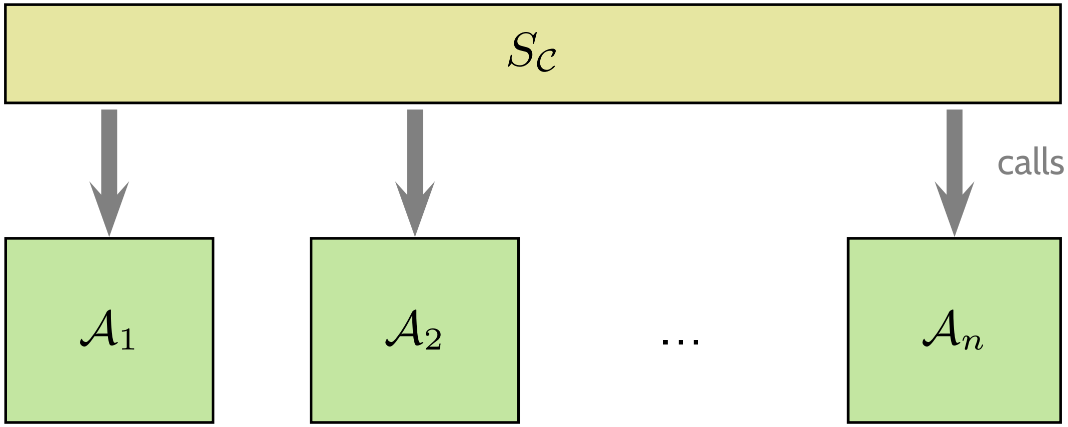
- Now that we have a CFA **model** $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ (**thoroughly checked** using Uppaal), we would like to have **executable software** – an implementation of the model.

- This task can be split into two sub-tasks:

  (i) **implement** each **CFA** $\mathcal{A}_i$ in the model by module $S_{\mathcal{A}_i}$,

  (ii) **implement** the **communication** in the network by module $S_\mathcal{C}$.

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



- **Fully distributed implementation** (without $S_\mathcal{C}$): different story, possible for sub-class of CFA.

Communication / Synchronisation

- Let $\mathcal{N} = \mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ with **pairwise disjoint variables**.
- Assume $B = B_{input} \dot\cup B_{internal}$, where B_{input} are **dedicated input channels**, i.e. there is no edge with action $a!$ and $a \in B_{input}$.
- Then software $S_{\mathcal{N}}$ consists of $S_{\mathcal{A}_1}, \dots, S_{\mathcal{A}_n}$ and the following $S_{\mathcal{C}}$.

$Set\langle Act\rangle\ R_1 := R_{1,ini}, \dots, R_n := R_{n,ini}$; // *initially enabled actions*

void main() { //snapshot

do

$\square$ *true* : $(\alpha, snd, rcv) := select(R_1, \dots, R_n)$; // *choose synchronisation*

// $(rcv = 0$ *if* $\alpha = \tau$,

// *blocks on deadlock*)

for $(k = 1$ **to** $n)$

if

for (k=1 to n) if

$~~\square~~$ $snd = k$: $R_k := \underline{take\_action_k}(\alpha)$ // *sender*

$~~\square~~$ $rcv = k$: $R_k := take\_action_k(\bar\alpha)$ // *receiver*

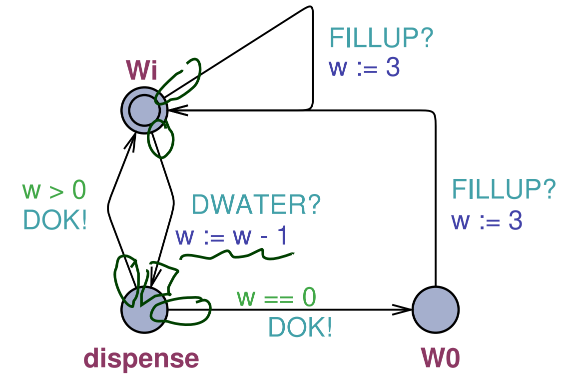
~~**fi**~~

// *snapshot*

od

}

Example



$$
\begin{array}{l}
\textit{int } w \;:=\; 3; \\[4pt]
\textbf{\textit{typedef}}\ \{\mathit{Wi}, \mathit{dispense}, \mathit{W0}\}\ \mathit{st\_T}; \\
\mathit{st\_T}\ \mathit{st}\ :=\ \mathit{Wi}; \\[4pt]
\mathit{Set}\langle \mathit{Act}\rangle\ \mathit{take\_action}(\ \mathit{Act}\ \alpha\ )\ \{ \\
\quad \mathit{Set}\langle \mathit{Act}\rangle\ R\ :=\ \emptyset; \\
\quad \textbf{\textit{if}} \\
\quad \square\ \mathit{st} = \mathit{Wi} : \quad \textbf{\textit{if}} \\
\qquad \square\ \alpha = \mathit{DWATER?} : \quad w := w - 1; \\
\qquad\qquad \mathit{st} := \mathit{dispense}; \\
\qquad\qquad \textbf{\textit{if}}\ (w = 0)\ \ R := R \cup \{\mathit{DOK!}\}; \\
\qquad\qquad \textbf{\textit{if}}\ (w > 0)\ \ R := R \cup \{\mathit{DOK!}\}; \\
\qquad \square\ \alpha = \mathit{FILLUP?} : \quad w := 3; \\
\qquad\qquad \mathit{st} := \mathit{Wi}; \\
\qquad\qquad R := R \cup \{\mathit{FILLUP?}, \mathit{DWATER?}\}; \\
\qquad \textbf{\textit{fi}}; \\
\quad \square\ \mathit{st} = \mathit{dispense} : \quad \textbf{\textit{if}} \\
\qquad \square\ \alpha = \mathit{DOK!} \wedge w = 0 : \quad \mathit{st} := \mathit{W0}; \\
\qquad\qquad R := R \cup \{\mathit{FILLUP?}\}; \\
\qquad \square\ \alpha = \mathit{DOK!} \wedge w > 0 : \quad \mathit{st} := \mathit{Wi}; \\
\qquad\qquad R := R \cup \{\mathit{FILLUP?}\}; \\
\qquad \textbf{\textit{fi}}; \\
\quad \square\ \mathit{st} = \mathit{W0} : \quad \textbf{\textit{if}} \\
\qquad \square\ \alpha = \mathit{FILLUP?} : \quad w := 3; \\
\qquad\qquad \mathit{st} := \mathit{Wi}; \\
\qquad\qquad R := R \cup \{\mathit{FILLUP?}, \mathit{DWATER?}\}; \\
\qquad \textbf{\textit{fi}}; \\
\quad \textbf{\textit{fi}}; \\
\quad \textbf{\textit{return}}\ R; \\
\}
\end{array}
$$

Translation Scheme...

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

$$E = \{(\ell_1, \alpha_{1,1}, \varphi_{1,1}, \vec{r}_{1,1}, \ell'_{1,1}), \dots, (\ell_1, \alpha_{1,n_1}, \varphi_{1,n_1}, \vec{r}_{1,n_1}, \ell'_{1,n_1}),$$
$$\dots$$
$$(\ell_m, \alpha_{m,1}, \varphi_{m,1}, \vec{r}_{m,1}, \ell'_{m,1}), \dots, (\ell_m, \alpha_{m,n_m}, \varphi_{m,n_m}, \vec{r}_{m,n_m}, \ell'_{m,n_m})\} :$$

```
T1 v1 := v1,ini; ... Tk vk := vk,ini;

typedef {ℓ1, ..., ℓm} st_T;
st_T st := ℓini;

Set⟨Act⟩ take_action( Act α ) {
    Set⟨Act⟩ R := ∅;
    if
    ⋮
    □ st = ℓi : if
                ⋮
                □ α = αi,j ∧ φi,j : r⃗i,j;
                                    st := ℓ'i,j;
                                    if (ℓ'i,j = ℓ1 ∧ φ1,1) R := R ∪ {α1,1};
                                    ⋮
                                    if (ℓ'i,j = ℓm ∧ φm,nm) R := R ∪ {αm,nm};
                ⋮
                fi;
    ⋮
    fi;
    return R;
}
```

$(\ell_i, \alpha_{i,j}, \varphi_{i,j}, \vec{r}_{i,j}, \ell'_{i,j})$

Deterministic CFA

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

$$\forall\, c \in \mathit{Conf}(\mathcal{C}) \text{ reachable } \forall\, \lambda \in B_{!?} \cup \{\tau\}\; \forall\, c_1, c_2 \in \mathit{Conf}(\mathcal{C}) \bullet$$
$$c \xrightarrow{\lambda} c_1 \wedge c \xrightarrow{\lambda} c_2 \implies c_1 = c_2 .$$

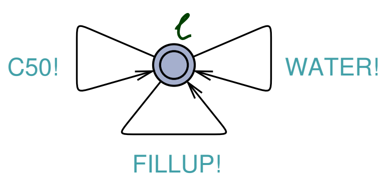
Proposition. Whether $\mathcal{C}$ is deterministic is **decidable**.

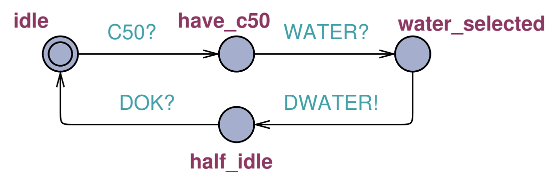
Proposition. If $\mathcal{C}$ is deterministic, then the translation of $\mathcal{C}$ is a **deterministic program**.

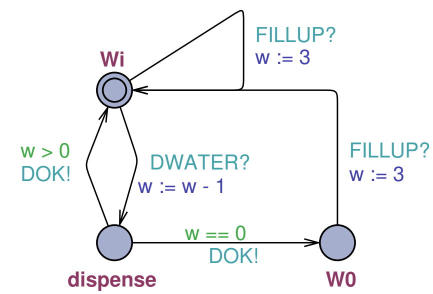
Model vs. Implementation

- Define $[\![S_\mathcal{N}]\!]$ to be the set of computation paths $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$

  such that σ_i has the values at '*snapshot*' at the i-th iteration and α_i is the i-th action.
- Then $[\![S_\mathcal{N}]\!]$ **bisimulates** the behaviour $[\![\mathcal{C}]\!]$ of model $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$.







$J(\ell): \sigma_0 \xrightarrow{C50} \sigma_1 \xrightarrow{WATER} \sigma_2 \xrightarrow{DWATER} \sigma_3 \cdots$

$w=3$

$(\ell, idle, W_i)$

$[\![S_C]\!]: \tilde{\sigma}_{0,1} \to \cdots \to \tilde{\sigma}_{0,n_1} \to \cdots \to \sigma_{1,n_1} \to \cdots$

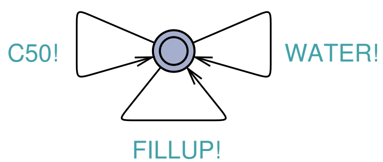
//snapshot $w = 3$ $st = W_i$

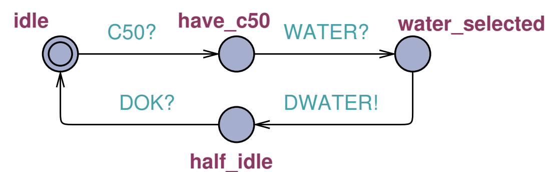
/snapshot $w=3$ $st = W_i$

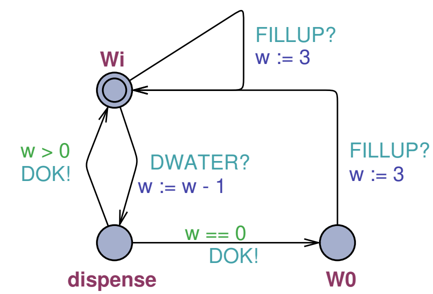
Model vs. Implementation

- Define $[\![S_{\mathcal{N}}]\!]$ to be the set of computation paths $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$

  such that σ_i has the values at '$snapshot$' at the i-th iteration and α_i is the i-th action.
- Then $[\![S_{\mathcal{N}}]\!]$ **bisimulates** the behaviour $[\![\mathcal{C}]\!]$ of model $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$.





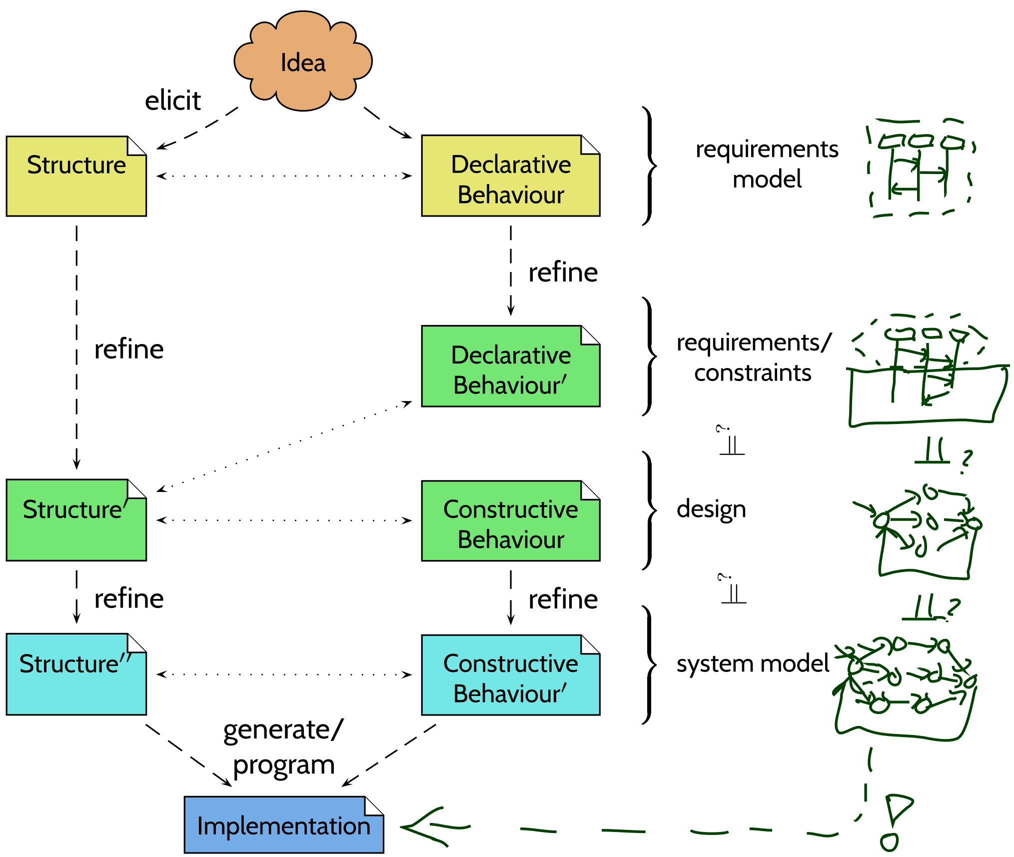


- Yes, and...?
  - If Uppaal reports that $\mathcal{N}_{\mathrm{VM}} \models \exists\lozenge\, w = 0$ holds, then $w = 0$ **(should be) reachable** in $[\![S_{\mathcal{N}_{\mathrm{VM}}}]\!]$.
  - If Uppaal reports that $\mathcal{N}_{\mathrm{VM}} \models \forall\square$ `tea_enabled imply` CoinValidator.`have_c150` holds, then $[\![S_{\mathcal{N}_{\mathrm{VM}}}]\!]$ **(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**.

Model-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.

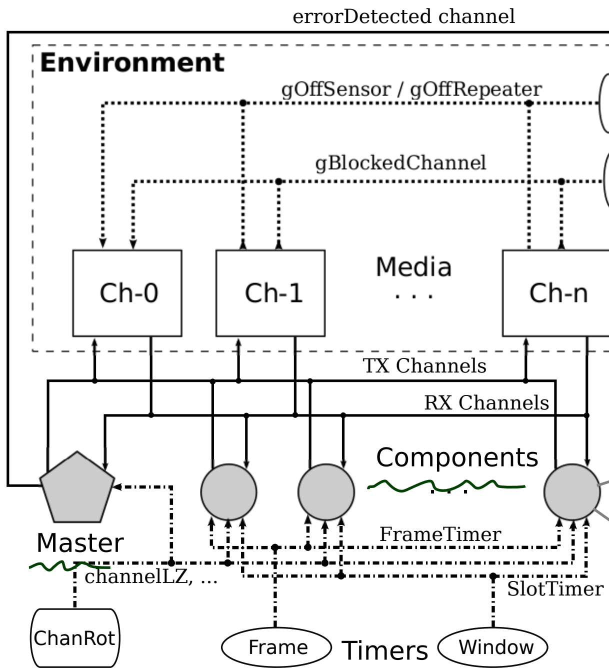


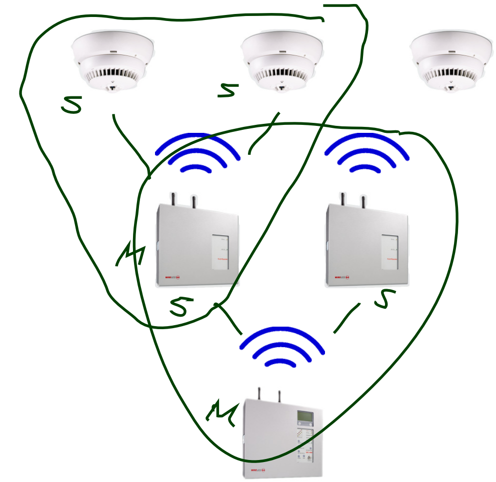
Content I (Architecture & Design)

- **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

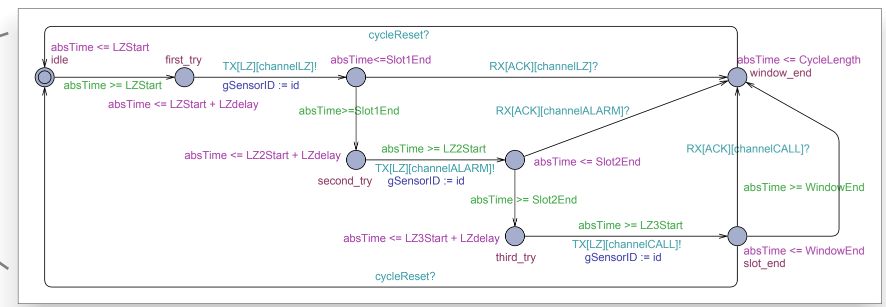
Formal Methods in the Real World: Case Study

Case Study: Wireless Fire Alarm System





(Arenis et al., 2014)



(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 [...].

$$\bigwedge_{i\in C}\Box\,(\lceil \mathit{FAIL} = i \wedge \neg \mathit{DET}_i \rceil \implies \ell \leq 300\mathrm{s})$$

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

$$\bigwedge_{i\in C}\lceil\, \overline{\mathit{ALARM}_{\{i\}}}\,\rceil \implies \Box\,(\lceil \mathit{ALARM}_i \wedge \neg \mathit{DISP}_i \rceil \implies \ell \leq 10\mathrm{s})\,,$$

Figures (Arenis et al., 2016)

Monitoring	Templates	Instances	Total Locations	Clocks
Sensors as slaves	9	137	1040	6
Repeaters as slaves	9	21	82	6

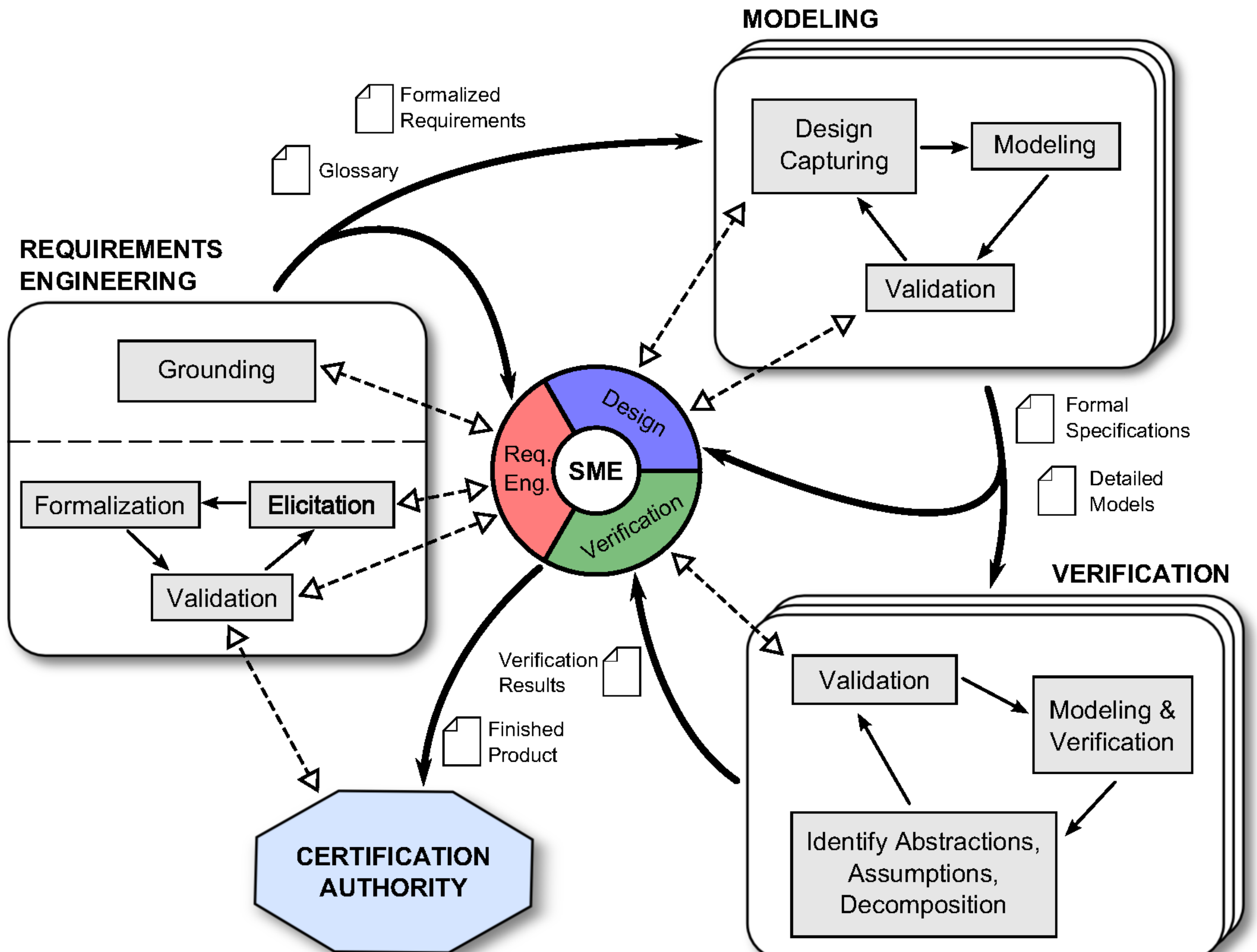
		Sensors as slaves, $N = 126$.			Repeaters as slaves, $N = 10$.		
	Query	**seconds**	**MB**	**States explored**	**seconds**	**MB**	**States explored**
Q1	Detection possible `E<> switcher.DETECTION`	10,205.13	557.00	26,445,788	38.21	55.67	1,250,596
Q2	No message collision `A[] not deadlock`	12,895.17	2,343.00	68,022,052	368.58	250.91	9,600,062
Q3	Detect$_T$ `A[] (switcher.DETECTION imply switcher.timer <= 300*Second)`	36,070.78	3,419.00	190,582,600	231.84	230.59	6,009,120
Q4	NoSpur$_T$ `A[] !center.ERROR`	97.44	44.29	640,943	3.94	10.14	144,613

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

	Model sequential	**Model optimized**	**Model test scenario**	**Measured Avg.**
First Alarm	$3.26s$	$2.14s$	$3.31s$	$2.79s \pm 0.53s$
All 10 Alarms	$29.03s$	$27.08s$	$29.81s$	$29.65s \pm 3.26s$

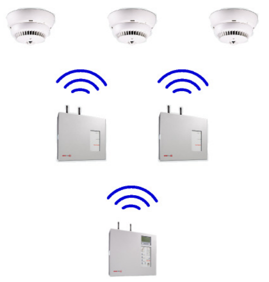
Predicted alarm transmission times vs. Measurements on real hardware.

Process Model

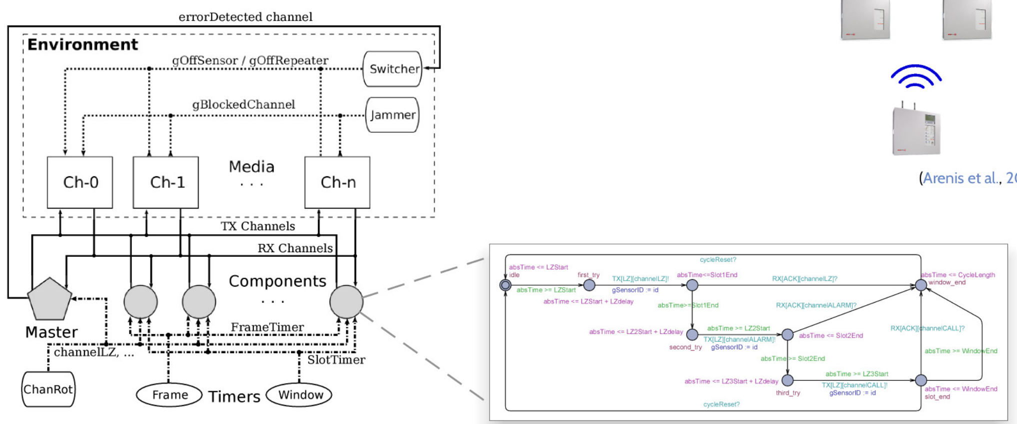


Advertisement

Case Study: Wireless Fire Alarm System



(Arenis et al., 2014)



(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** [...].

$$\bigwedge_{i\in C}\Box\,(\lceil \mathit{FAIL} = i \wedge \neg \mathit{DET}_i \rceil \implies \ell \le 300\mathrm{s})$$

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

$$\bigwedge_{i\in C}\lceil\, \overline{\mathit{ALARM}_{\{i\}}}\, \rceil \implies \Box\,(\lceil \mathit{ALARM}_i \wedge \neg \mathit{DISP}_i \rceil \implies \ell \le 10\mathrm{s})\,,$$

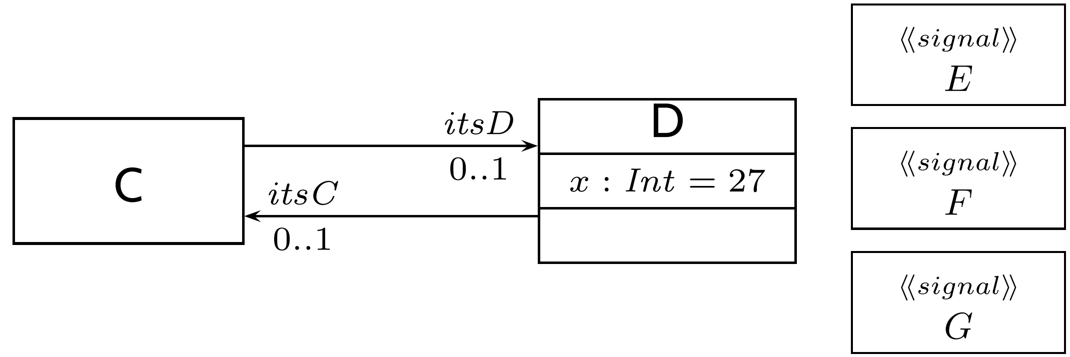
$\to$ Lecture "**Real-Time Systems**" in Winter 2017/18.

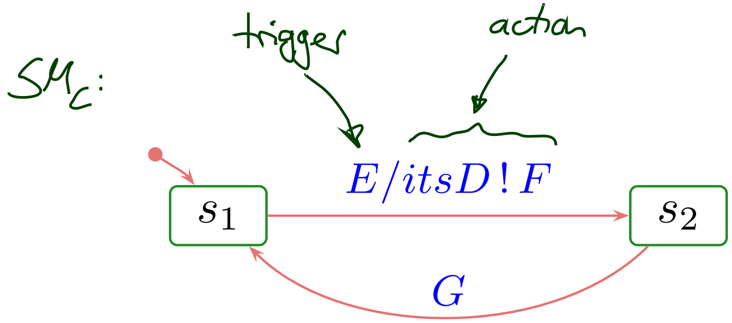
Content I (Architecture & Design)

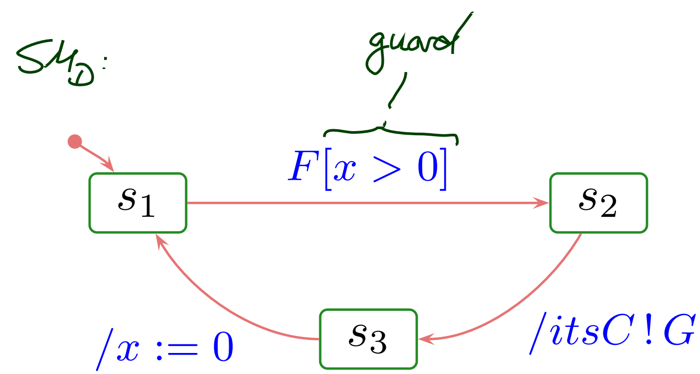
- **CFA vs. Software**
  - a CFA model is software
  - **implementing** CFA
  - formal methods in the real world: case study
- **UML State Machines**
  - **Core** State Machines
  - steps and run-to-completion steps
  - **Hierarchical State Machines**
  - **Rhapsody**
- **Unified Modelling Language**
  - Brief History
  - **Sub-Languages**
  - UML Modes

UML State Machines

UML Core State Machines





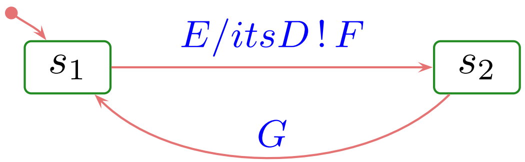


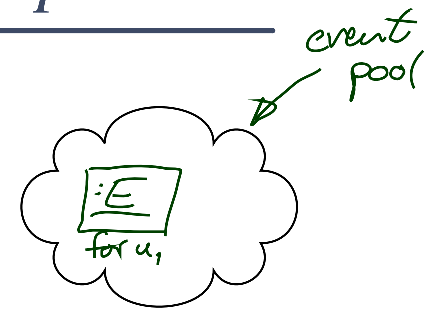
$$annot ::= \Big[\,\underbrace{\langle event\rangle[\,\mathbf{.}\,\langle event\rangle]^*}_{trigger}\quad [\,\mathbf{[}\,\langle guard\rangle\,\mathbf{]}\,]\quad [\,\mathbf{/}\,\langle action\rangle]\,\Big]$$

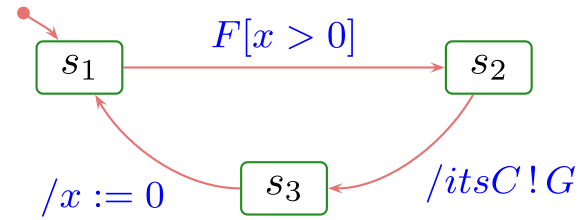
with

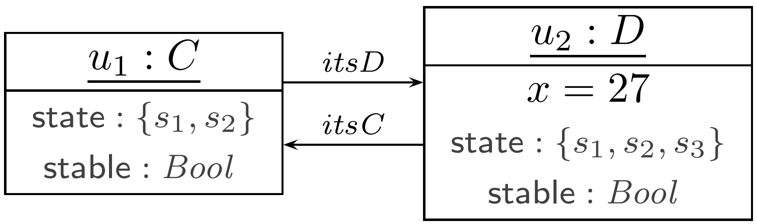
- $event \in \mathcal{E}$, (optional)
- $guard \in Expr_{\mathscr{S}}$ (default: *true*, assumed to be in $Expr_{\mathscr{S}}$)
- $action \in Act_{\mathscr{S}}$ (default: `skip`, assumed to be in $Act_{\mathscr{S}}$)

Event Pool and Run-To-Completion



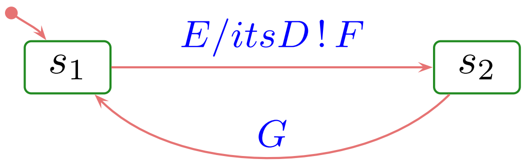


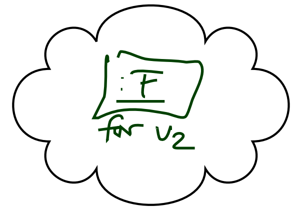


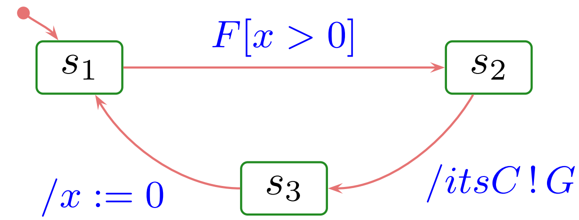


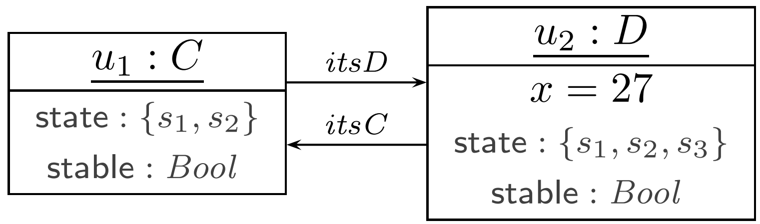
	u_1		u_2			
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1

Event Pool and Run-To-Completion



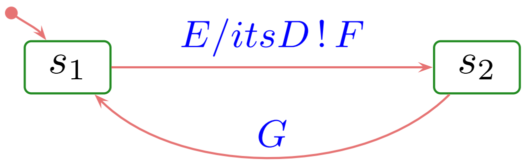


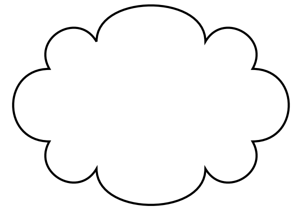


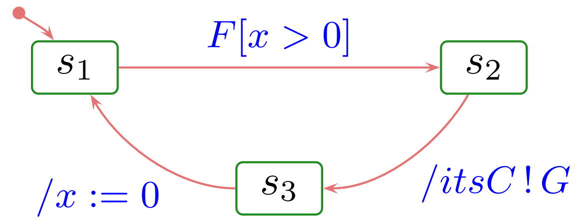


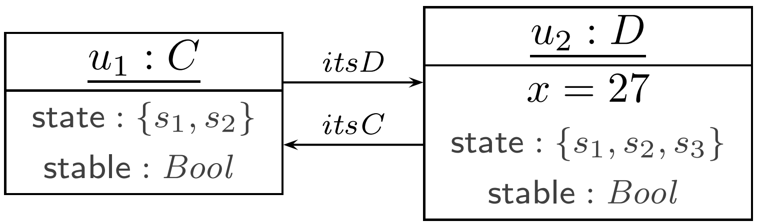
	u_1		u_2			
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2

Event Pool and Run-To-Completion



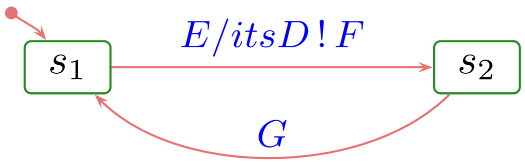


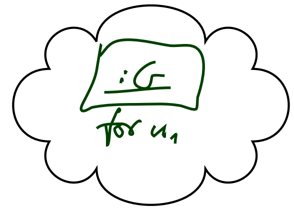


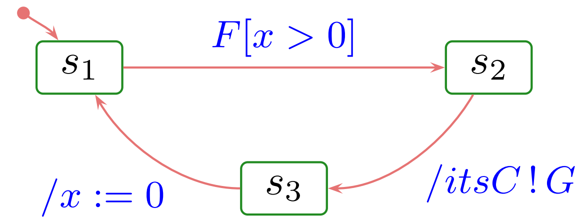


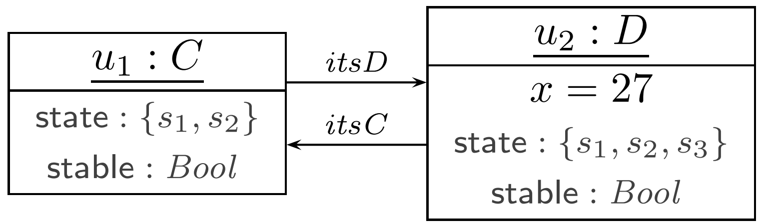
	u_1		u_2			
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	

Event Pool and Run-To-Completion



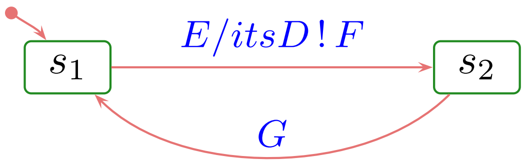


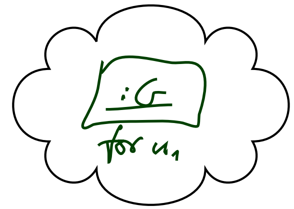


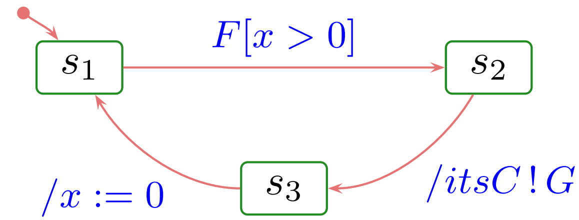


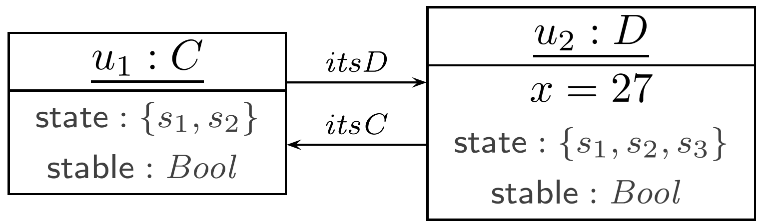
	u_1		u_2			
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1

Event Pool and Run-To-Completion



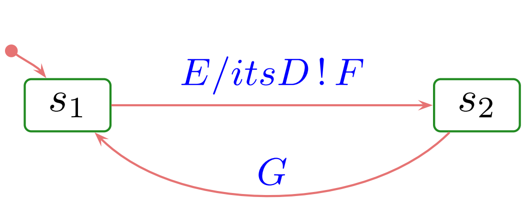


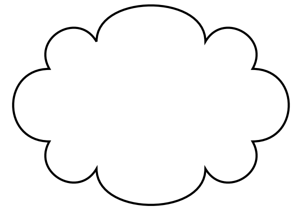


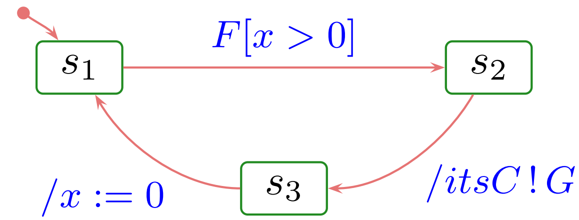


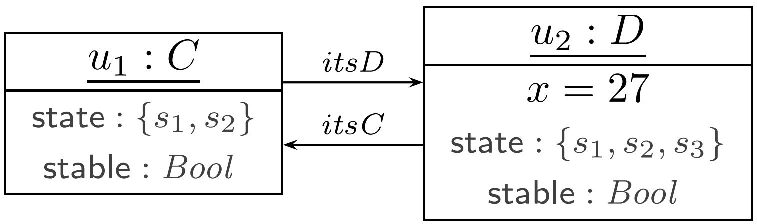
	u_1		u_2			
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1
4.a	s_2	1	0	s_1	1	G ready for u_1

Event Pool and Run-To-Completion



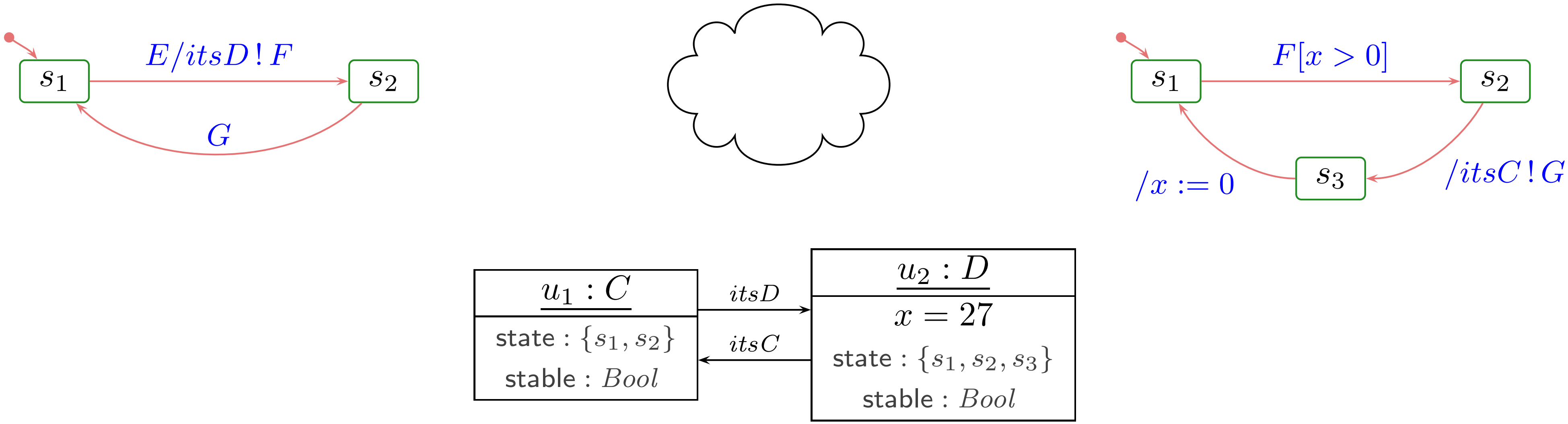






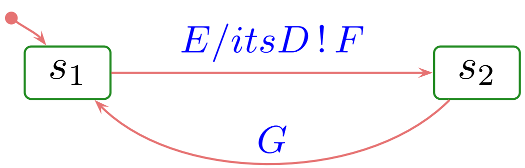
	u_1		u_2			
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1
4.a	s_2	1	0	s_1	1	G ready for u_1
5.a	s_1	1	0	s_1	1	

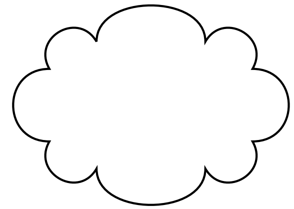
Event Pool and Run-To-Completion

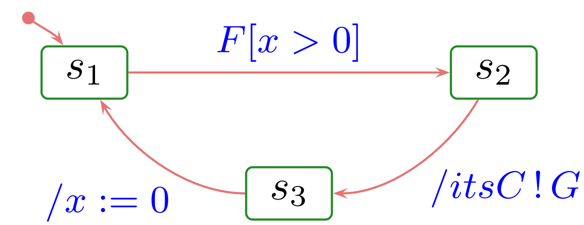


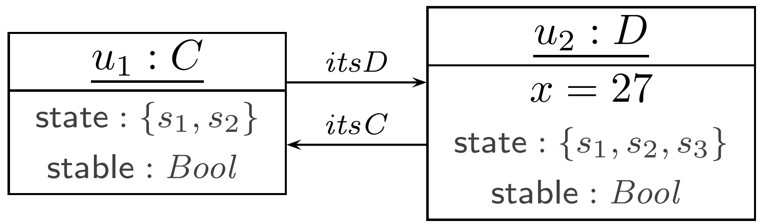
	u_1		u_2			
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1
4.a	s_2	1	0	s_1	1	G ready for u_1
5.a	s_1	1	0	s_1	1	
4.b	s_1	1	27	s_3	0	

Event Pool and Run-To-Completion



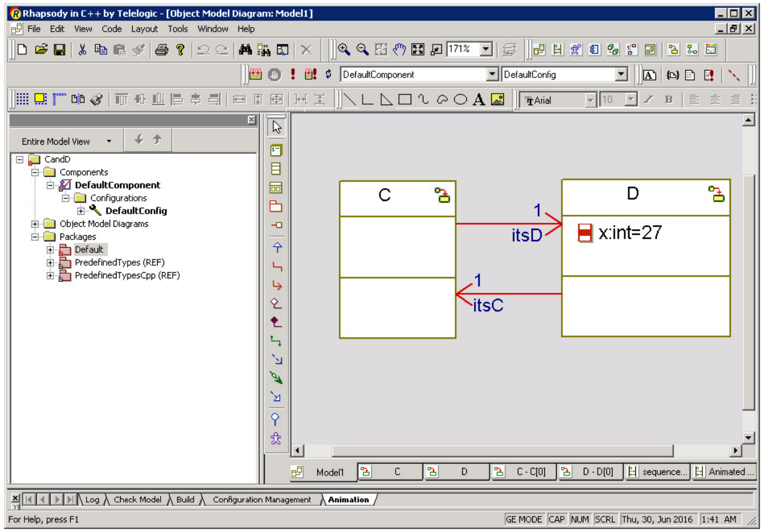




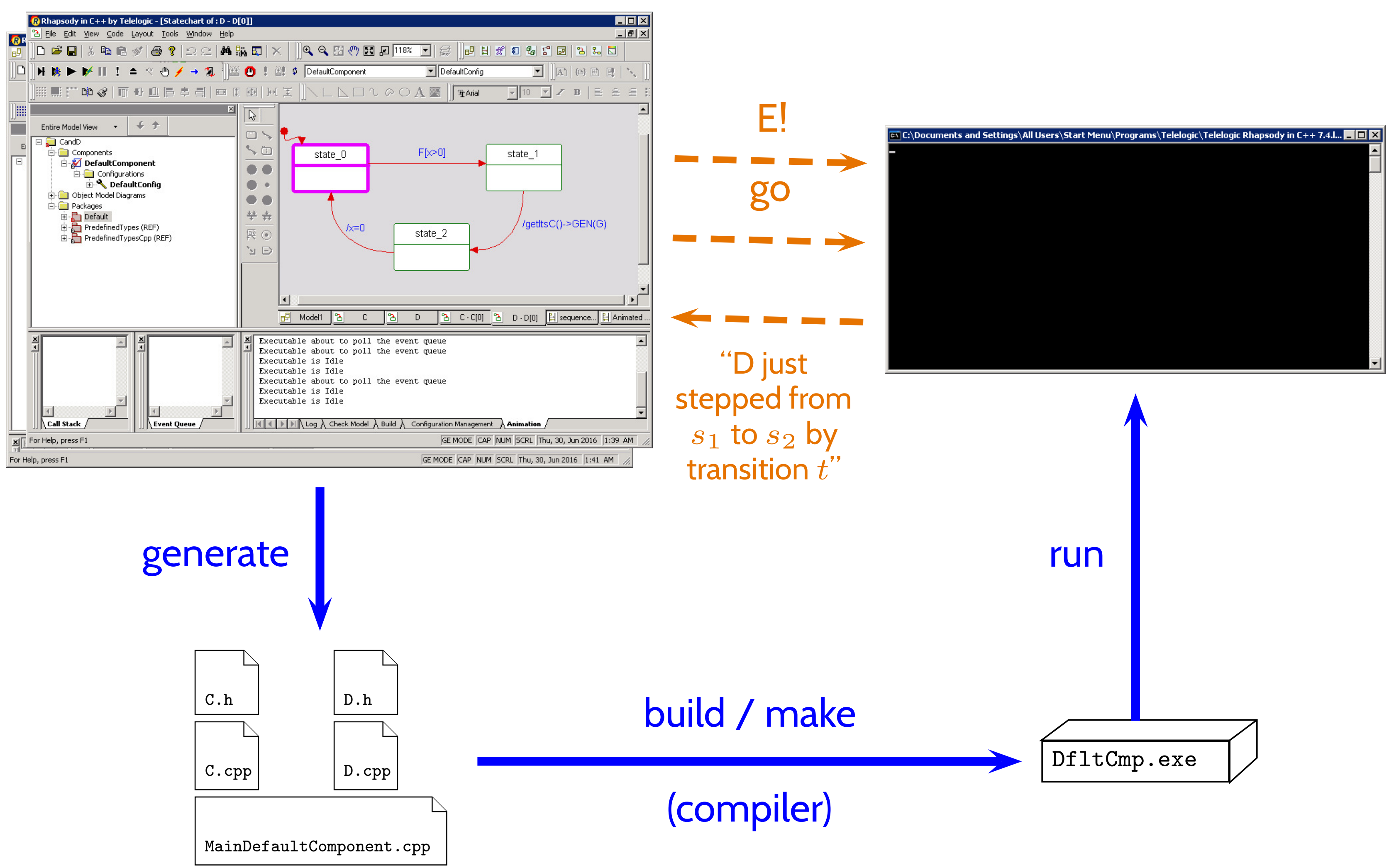


	u_1			u_2		
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1
4.a	s_2	1	0	s_1	1	G ready for u_1
5.a	s_1	1	0	s_1	1	
4.b	s_1	1	27	s_3	0	
5.b	s_1	1	0	s_1	1	

Rhapsody Architecture

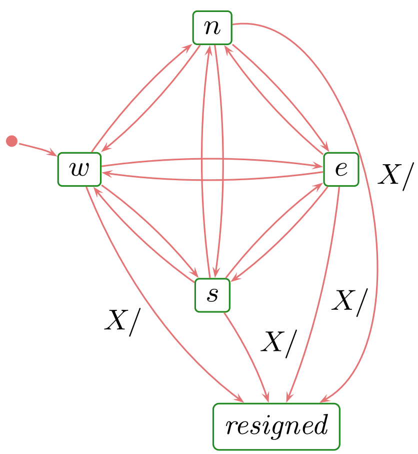


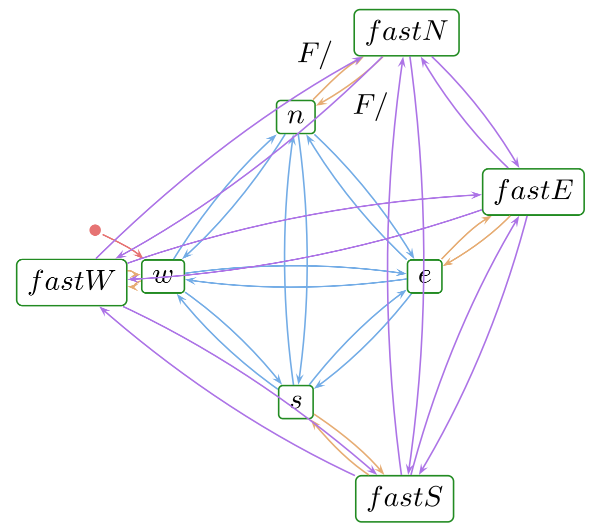
Rhapsody Architecture



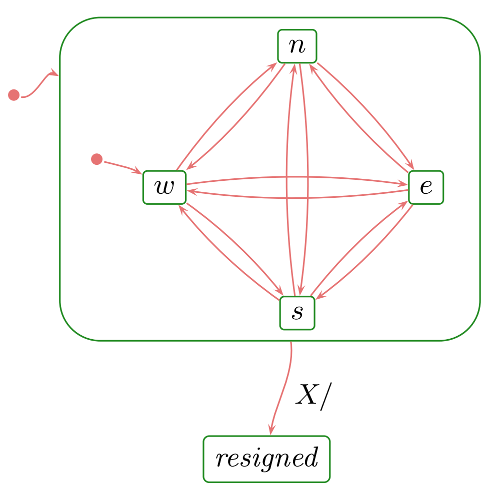
Composite (or Hierarchical) States

- OR-states, AND-states Harel (1987).
- Composite states are about **abbreviation**, **structuring**, and **avoiding redundancy**.

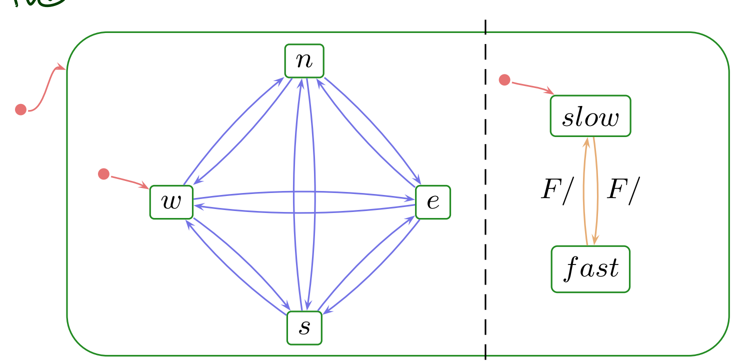




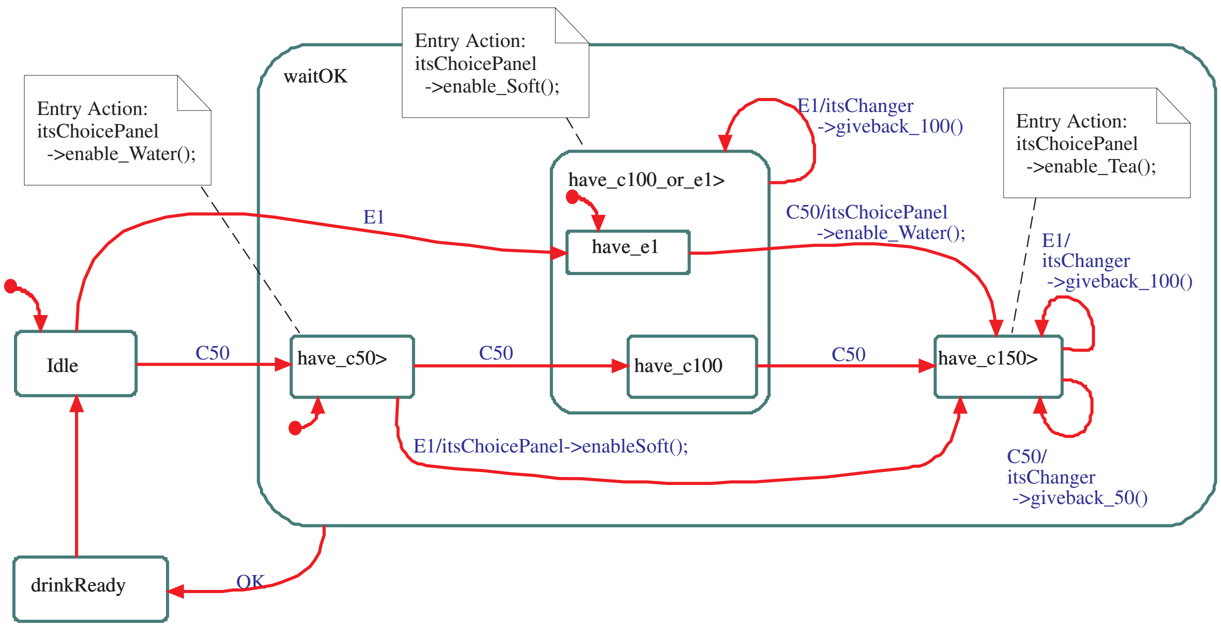
OR:



AND:

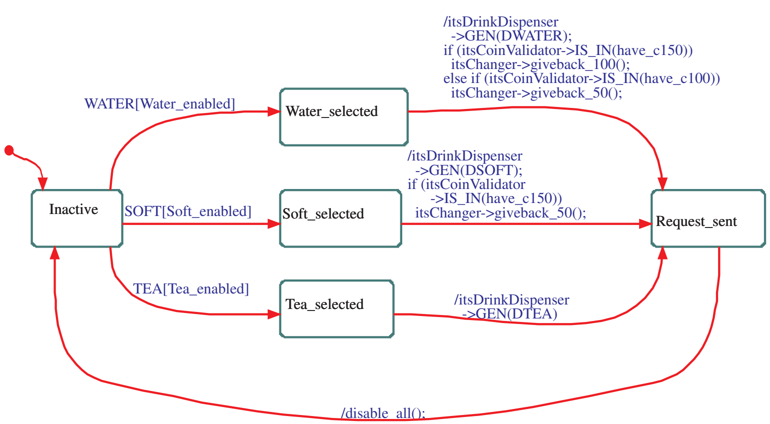


Example

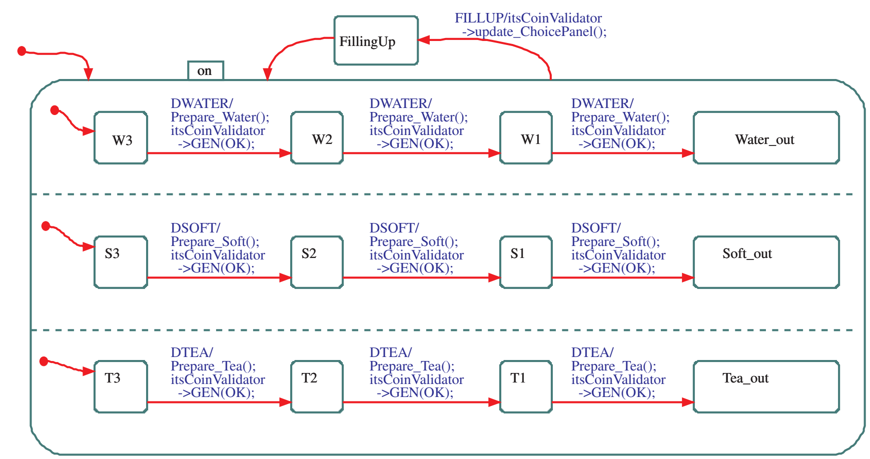


waitOK
Entry Action:
itsChoicePanel
->enable_Water();
Entry Action:
itsChoicePanel
->enable_Soft();
Entry Action:
itsChoicePanel
->enable_Tea();
E1/itsChanger
->giveback_100()
have_c100_or_e1>
have_e1
E1
C50/itsChoicePanel
->enable_Water();
E1/
itsChanger
->giveback_100()
Idle
C50
have_c50>
C50
have_c100
C50
have_c150>
E1/itsChoicePanel->enableSoft();
C50/
itsChanger
->giveback_50()
drinkReady
OK

STUDENTENWERK
OLDENBURG
80
WATER
SOFT
TEA

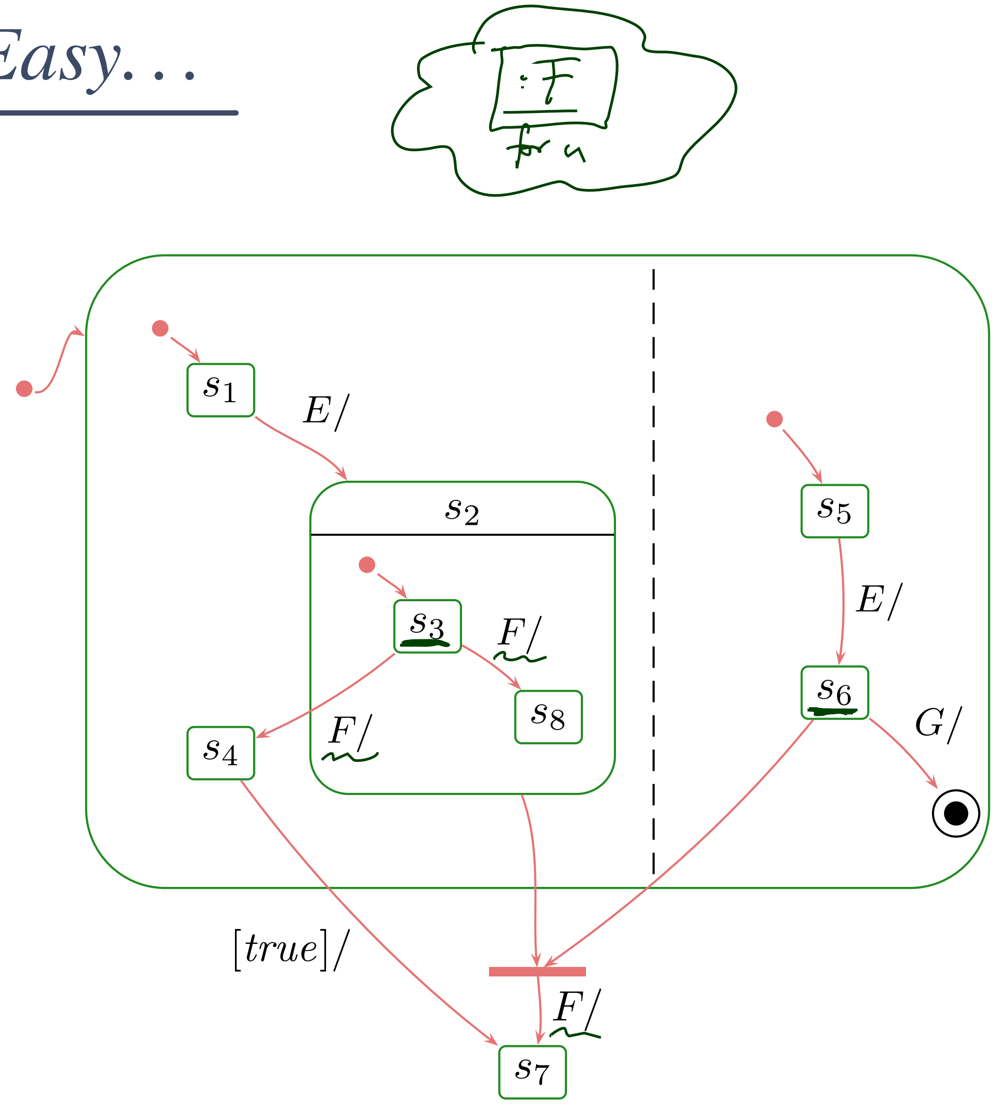


/itsDrinkDispenser
->GEN(DWATER);
if (itsCoinValidator->IS_IN(have_c150))
itsChanger->giveback_100();
else if (itsCoinValidator->IS_IN(have_c100))
itsChanger->giveback_50();
WATER[Water_enabled]
Water_selected
/itsDrinkDispenser
->GEN(DSOFT);
if (itsCoinValidator
->IS_IN(have_c150))
itsChanger->giveback_50();
Inactive
SOFT[Soft_enabled]
Soft_selected
Request_sent
TEA[Tea_enabled]
Tea_selected
/itsDrinkDispenser
->GEN(DTEA)
/disable_all();



FillingUp
FILLUP/itsCoinValidator
->update_ChoicePanel();
on
W3
DWATER/
Prepare_Water();
itsCoinValidator
->GEN(OK);
W2
W1
Water_out
S3
DSOFT/
Prepare_Soft();
itsCoinValidator
->GEN(OK);
S2
S1
Soft_out
T3
DTEA/
Prepare_Tea();
itsCoinValidator
->GEN(OK);
T2
T1
Tea_out

Would be Too Easy...



$\rightarrow$ "**Software Design, Modelling, and Analysis with UML**" in some winter semesters.

Content I (Architecture & Design)

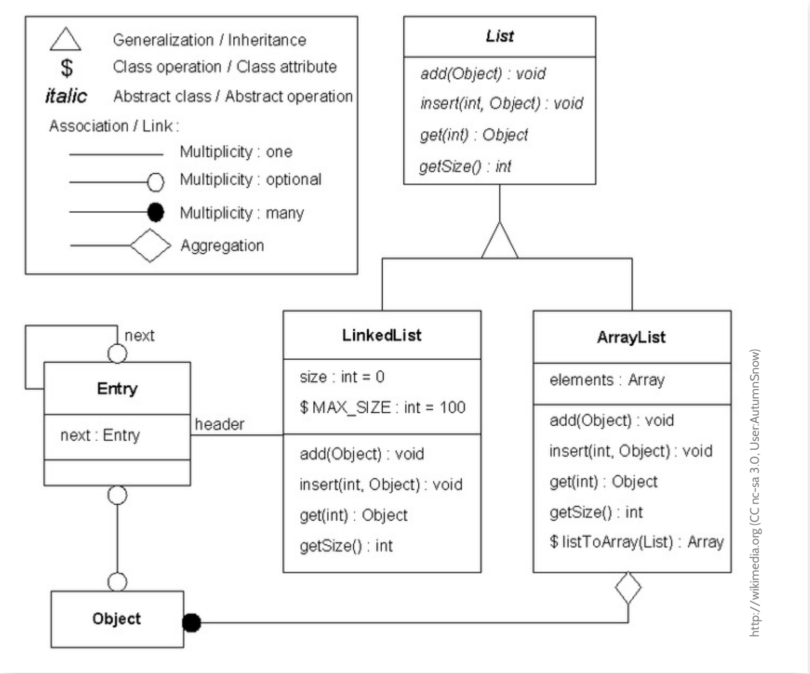
- **CFA vs. Software**
  - a CFA model is software
  - **implementing** CFA
  - formal methods in the real world: case study
- **UML State Machines**
  - **Core** State Machines
  - steps and run-to-completion steps
  - **Hierarchical State Machines**
  - **Rhapsody**
- **Unified Modelling Language**
  - Brief History
  - **Sub-Languages**
  - UML Modes

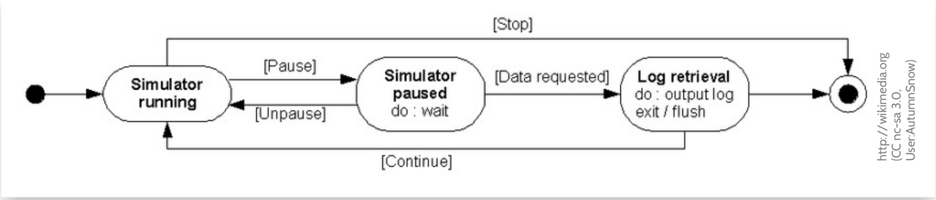
A Brief History of the Unified Modelling Language (UML)

- 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:

A Brief History of the Unified Modelling Language (UML)

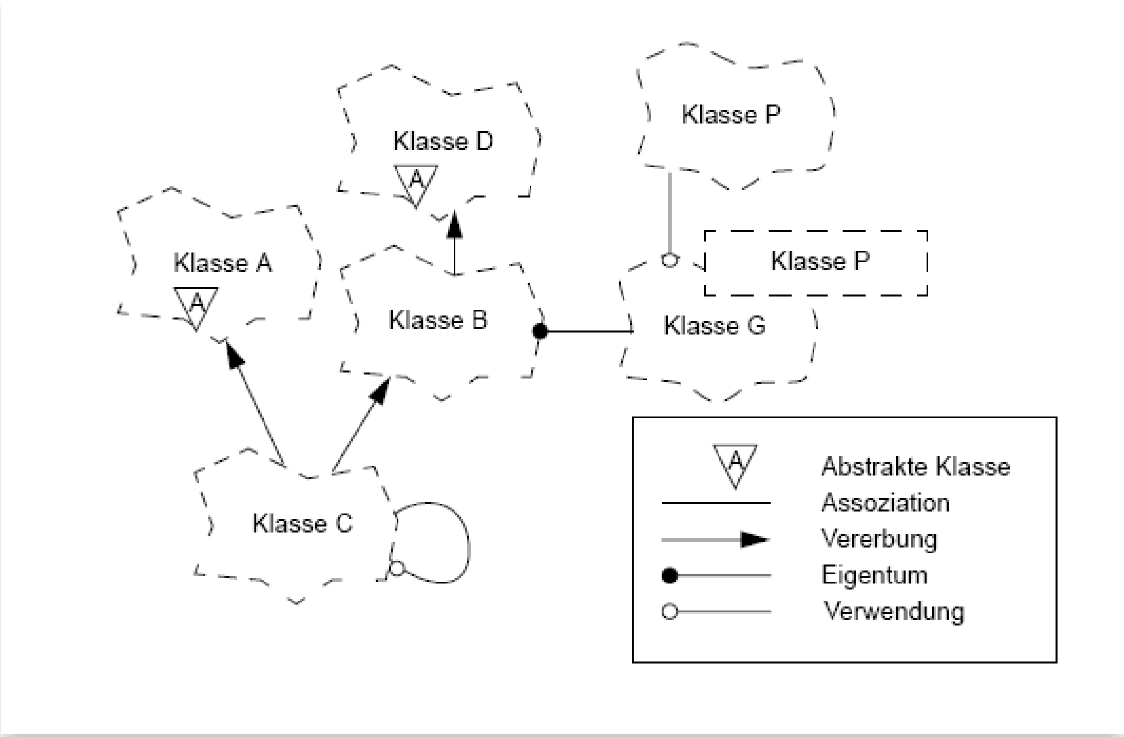
- Boxes/lines and finite automata are used to visualise software **for ages**.
- **1970's**, **Software Crisis**™
  – Idea: learn from engineering disciplines to handle growing complexity.
  Modelling languages: **Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams**
- Mid **1980**'s: **Statecharts** (Harel, 1987), **StateMate**™ (H
- Early **1990's**, advent of **Object-Oriented**-Analysis/D
  – Inflation of notations and methods, most prominer
  - **Object-Modeling Technique** (OMT)
    (Rumbaugh et al., 1990)





A Brief History of the Unified Modelling Language (UML)

- 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, m
  - **Object-Modeling Technique** (OMT) (Rumbaugh et al., 1990)
  - **Booch Method and Notation** (Booch, 1993)

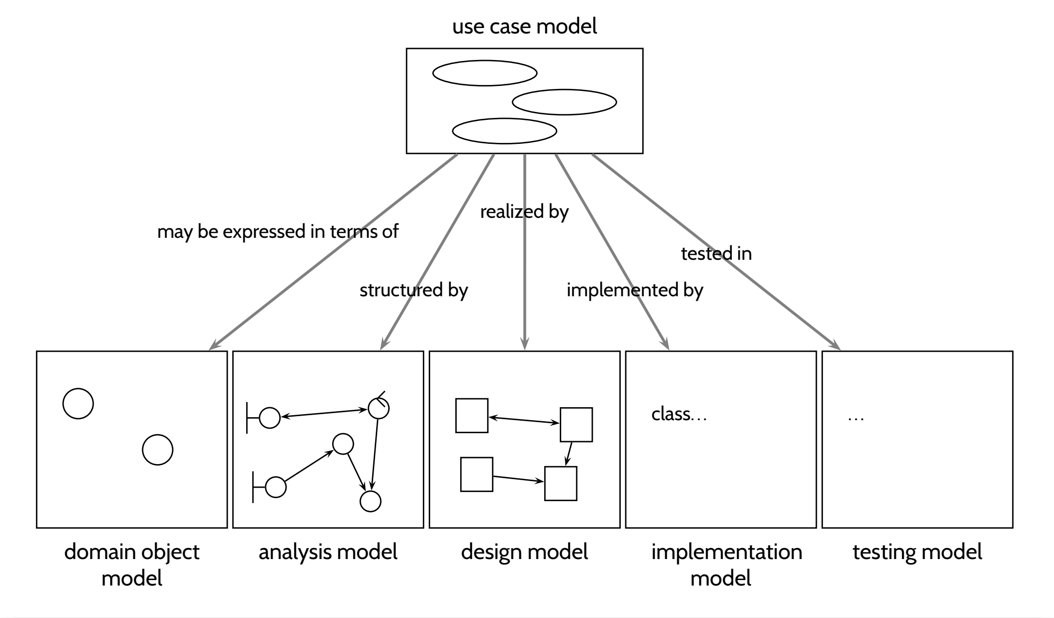


http://wikimedia.org (Public domain, Johannes Fasolt)

A Brief History of the Unified Modelling Language (UML)

- 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...



A Brief History of the Unified Modelling Language (UML)

- 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)

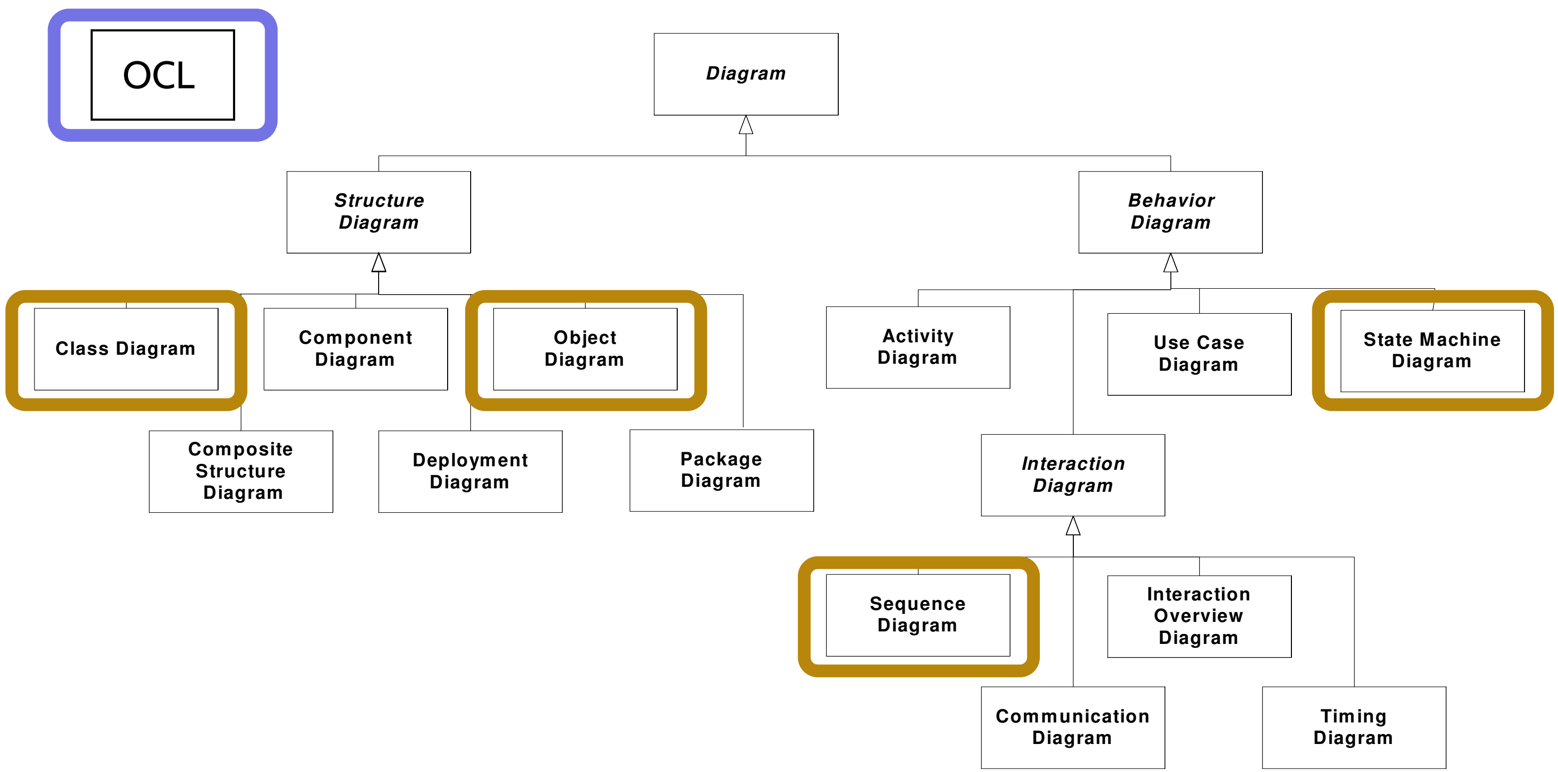


Figure A.5 - The taxonomy of structure and behavior diagram

Dobing and Parsons (2006)

UML Modes

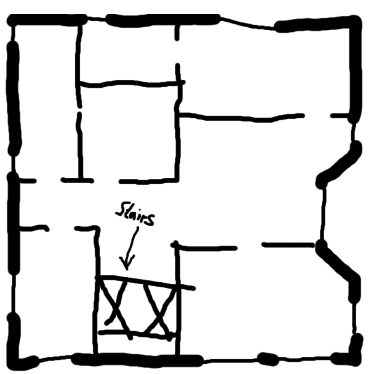
UML and the Pragmatic Attribute

Recall: definition "model" (Glinz, 2008, 425):

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

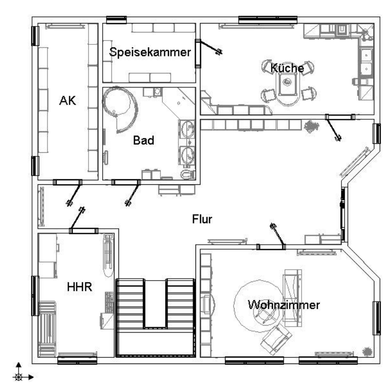
Examples for context/purpose:

Floorplan as sketch:



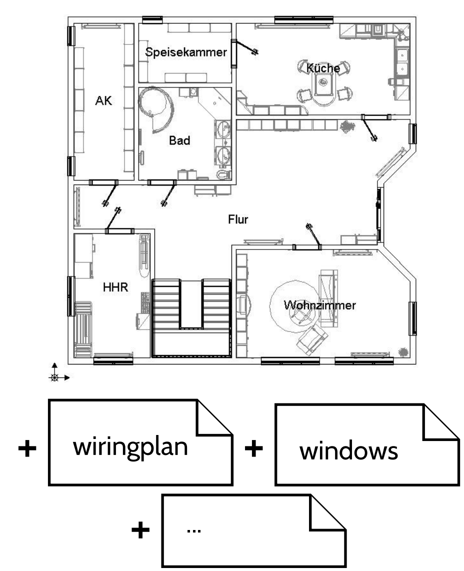
GameState → Segment
x, y, l, d

Floorplan as blueprint:



→ Segment
x: int
⋮

Floorplan as program:



With UML it's the Same [http://martinfowler.com/bliki]

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**.

With UML it's the Same [http://martinfowler.com/bliki]

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 ra- ther 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 "compre- hensiveness 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 sup- port 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
- but a

UML-Mode of the Lecture: As Blueprint

- 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

VL 14
⋮
VL 15
⋮
VL 16
⋮
VL 17
⋮

- **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**

Content (Part II)

- **Introduction**
  - quotes on testing,
  - systematic testing vs. 'rumprobieren.'
- **Test Case**
  - definition,
  - execution,
  - **positive** and **negative**.
- The **Specification** of a Software
- **Test Suite**
- More **Vocabulary**

Testing: Introduction

Quotes On Testing

"Testing is the execution of a program with the goal to discover errors."

(**G. J. Myers, 1979**)

≈ vorführen

? "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**)

≈ beweisen

"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)

Recall:

Definition. Software is a finite description S of a (possibly infinite) set $[\![S]\!]$ of (finite or infinite) computation paths of the form $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$ 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 $[\![\cdot]\!] : S \mapsto [\![S]\!]$ 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 Σ_{in} and Σ_{out} such that
$$\Sigma = \Sigma_{in} \times \Sigma_{out}.$$
- **Computation paths** are then of the form
$$\pi = \begin{pmatrix} \sigma_0^i \\ \sigma_0^o \end{pmatrix} \xrightarrow{\alpha_1} \begin{pmatrix} \sigma_1^i \\ \sigma_1^o \end{pmatrix} \xrightarrow{\alpha_2} \cdots$$
- We use $\pi \downarrow \Sigma_{in}$ to denote $\pi = \sigma_0^i \xrightarrow{\alpha_1} \sigma_1^i \xrightarrow{\alpha_2} \cdots$, i.e. the **projection** of π onto Σ_{in}.

Test Case

Definition. A **test case** T over Σ 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:

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

Examples:

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

$$T = (\underbrace{s = \texttt{"abc"}}_{In}, \underbrace{r = 3}_{Soll})$$

$$[\![s = \texttt{"abc"}]\!] = \{\sigma_0^i \xrightarrow{\tau} \sigma_1^i \mid \sigma_0(s) = \texttt{"abc"}\}, \quad [\![r = 3]\!] = \{\sigma_0 \xrightarrow{\tau} \sigma_1 \mid \sigma_1(r) = 3\},$$

Shorthand notation: $T = (\texttt{"abc"}, 3)$.

- "Call `strlen`() with string `"abc"`, expect return value 3."

Test Case

Definition. A **test case** T over Σ 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:

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

Examples:

- Test case for vending machine.

$$T = (C50, WATER; DWATER)$$

$$[\![C50, WATER]\!] = \{\sigma_0^i \xrightarrow{C50} \sigma_1^i \xrightarrow{\tau} \cdots \xrightarrow{\tau} \sigma_{j-1}^i \xrightarrow{WATER} \sigma_j^i\},$$

$$[\![DWATER]\!] = \{\sigma_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} \sigma_{k-1} \xrightarrow{DWATER} \sigma_k \mid k \leq 10\},$$

- "Send event $C50$ and any time later $WATER$, expect $DWATER$ after 10 steps the latest."

Test Case

Definition. A **test case** T over Σ 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:

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

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

$$\pi = \begin{pmatrix} \sigma_0^i \\ \sigma_0^o \end{pmatrix} \xrightarrow{\alpha_1} \begin{pmatrix} \sigma_1^i \\ \sigma_1^o \end{pmatrix} \xrightarrow{\alpha_2} \cdots$$

  from $[\![S]\!]$ is called **execution** of test case $(In, Soll)$ if and only if

  - there is $n \in \mathbb{N}$ such that $\sigma_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \sigma_n \downarrow \Sigma_{in} \in [\![In]\!]$.

    ("A prefix of π corresponds to an input sequence").

Execution π of test case T is called

- **successful** (or **positive**) if and only if $\pi \notin [\![Soll]\!]$.
  - 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 $\pi \in [\![Soll]\!]$.
  - Intuition: no error has been discovered.
  - Alternative: test item S **passed the test**.
  - Okay: "test passed".

Not Executing Test Cases

- Consider the test case

$$T = (\texttt{""}, 0)$$

  for procedure `strlen`.

  ("Empty string has length 0.")

- A tester observes the following software behaviour:

$$\pi = \underbrace{\{s \mapsto \texttt{NULL}, r \mapsto 0\}}_{=\sigma_0} \xrightarrow{\tau} \underbrace{\textit{program-abortion}}_{\sigma_1}$$

- 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 $\{T_1, \dots, T_n\}$.

- 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

References

Arenis, S. F., Westphal, B., Dietsch, D., Muñiz, M., and Andisha, A. S. (2014). The wireless fire alarm system: Ensuring conformance to industrial standards through formal verification. In Jones, C. B., Pihlajasaari, P., and Sun, J., editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *LNCS*, pages 658–672. Springer.

Arenis, S. F., Westphal, B., Dietsch, D., Muñiz, M., Andisha, A. S., and Podelski, A. (2016). Ready for testing: ensuring conformance to industrial standards through formal verification. *Formal Asp. Comput.*, 28(3):499–527.

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

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

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

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

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

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

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

OMG (2007a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-04.

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