Softwaretechnik / Software-Engineering

Lecture 14: UML State Machines

2016-06-30

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

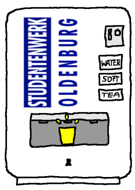
Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Architecture & Design: Content

VL 11	• **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)
VL 12	(iv) **modelling structure** a) (simplified) class diagrams b) (simplified) object diagrams c) (simplified) object constraint logic (OCL)
VL 13	(v) **modelling behaviour** a) communicating finite automata b) Uppaal query language
VL 14	c) implementing CFA d) an outlook on **UML State Machines**
VL 15	• **Design Patterns**
⋮	• **Testing**: Introduction

Content

Design Sanity Check: Drive to Configuration



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

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

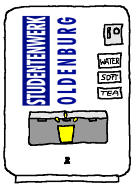
  $$w = 0$$

  is reachable, i.e. check

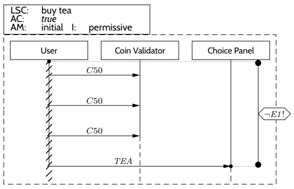
  $$\mathcal{N}_{\mathrm{VM}} \models \exists\Diamond\, w = 0.$$

  for the vending machine model $\mathcal{N}_{\mathrm{VM}}$.

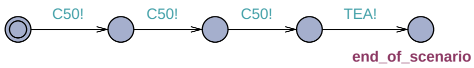
Design Check: Scenarios



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



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

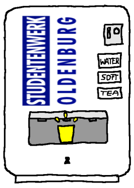


  instead of User and check whether location `end_of_scenario` is reachable, i.e. check

$$\mathcal{N}'_{\mathrm{VM}} \models \exists\Diamond\, \text{Scenario}.\texttt{end\_of\_scenario}.$$

  for the modified vending machine model $\mathcal{N}'_{\mathrm{VM}}$.

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.)

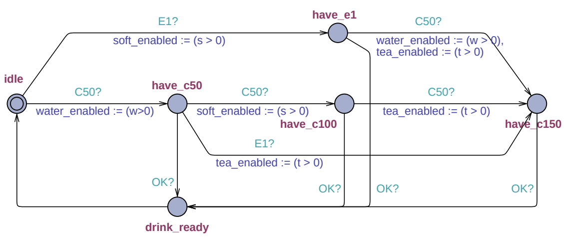
- **Approach**: Check whether the implication

$$\texttt{tea\_enabled} \implies \text{CoinValidator}.\texttt{have\_c150}$$

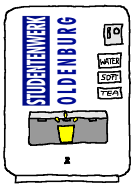
  holds in all reachable configurations, i.e. check

$$\mathcal{N}_{\mathrm{VM}} \models \forall\Box\, \texttt{tea\_enabled} \;\; \texttt{imply} \;\; \text{CoinValidator}.\texttt{have\_c150}$$

  for the vending machine model $\mathcal{N}_{\mathrm{VM}}$.



Design Verification: Sanity Check



- **Question**: Is the "tea" button **ever** enabled?
  (Otherwise, the considered invariant

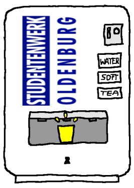
  $$\texttt{tea\_enabled} \implies \text{CoinValidator}.\texttt{have\_c150}$$

  holds vacuously.)

- **Approach**: Check whether a configuration satisfying $\texttt{water\_enabled} = 1$ is reachable.

  Exactly like we did with $w = 0$ earlier.

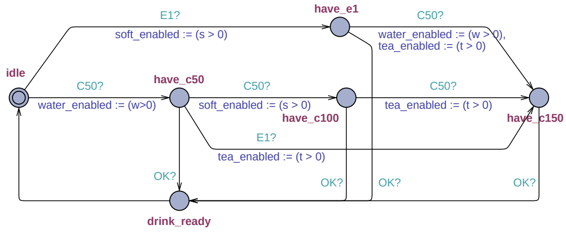
Design Verification: Another Invariant



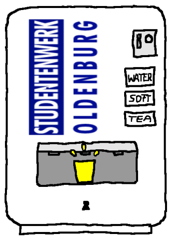
- **Question**: Is it the case that, if there is money in the machine and water in stock, that the "water" button is enabled?

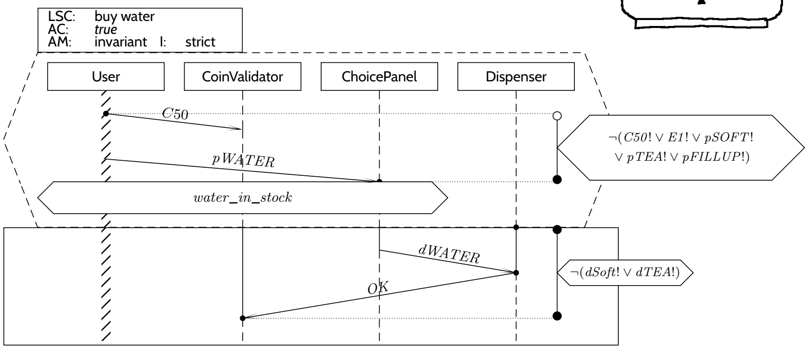
- **Approach**: Check

  $$\mathcal{N}_{\mathrm{VM}} \models \forall\Box\,(\text{CoinValidator}.\texttt{have\_c50}\ \texttt{or}\ \text{CoinValidator}.\texttt{have\_c100}\ \texttt{or}\ \text{CoinValidator}.\texttt{have\_c150})$$
  $$\texttt{imply}\ \texttt{water\_enabled}.$$

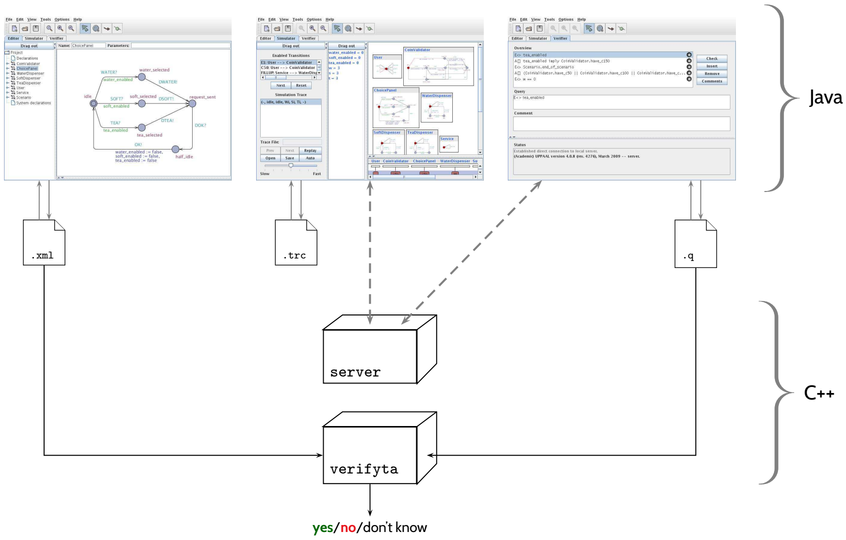


Recall: Universal LSC Example

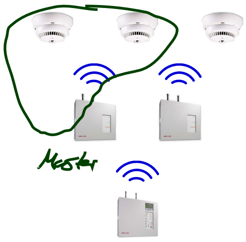




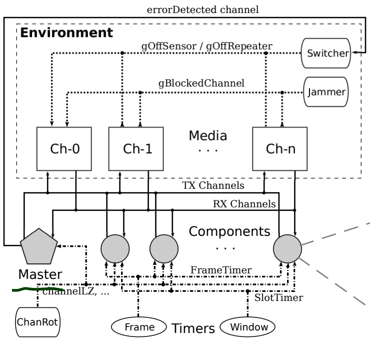
Uppaal Architecture

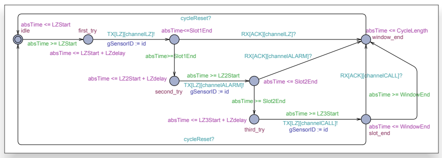


Case Study: Wireless Fire Alarm System



(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** [...].

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

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

$$\bigwedge_{i \in C} \lceil \overline{ALARM_{\{i\}}} \rceil \implies \Box(\lceil ALARM_i \wedge \neg DISP_i \rceil \implies \ell \leq 10\mathrm{s}),$$

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 $[\![S]\!]$ of (finite or infinite) computation paths of the form

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

where

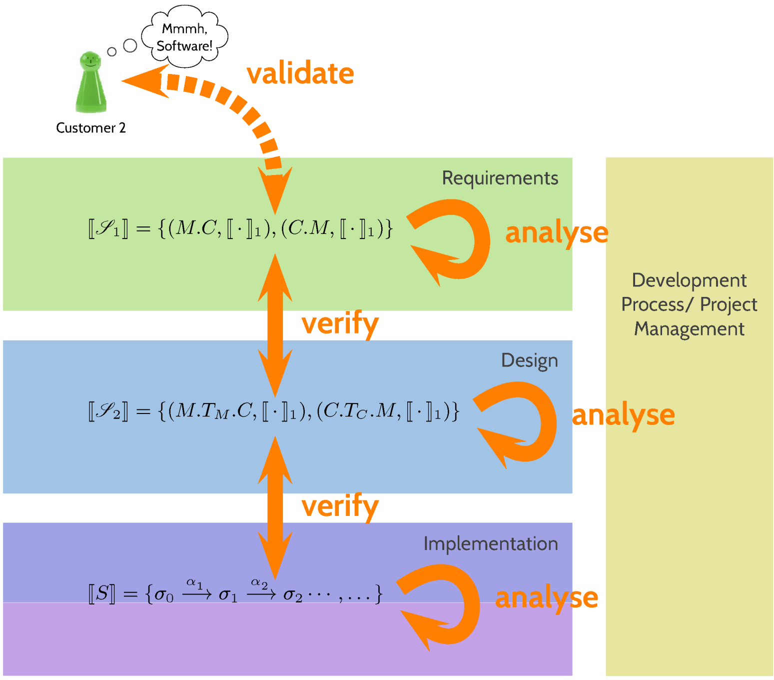
- $\sigma_i \in \Sigma$, $i \in \mathbb{N}_0$, is called **state** (or **configuration**), and
- $\alpha_i \in A$, $i \in \mathbb{N}_0$, is called **action** (or **event**).

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

- Let $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ 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 $\mathcal{C}\}$.

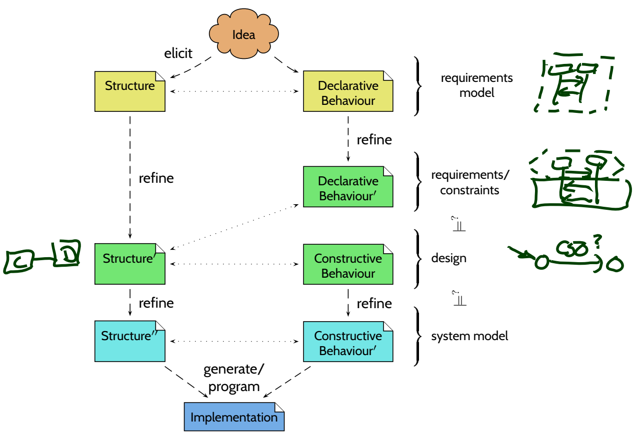
- **Note**: the structural model just consists of the set of variables and the locations of $\mathcal{C}$.

Formal Methods in the Software Development Process



Model-Driven Software Engineering

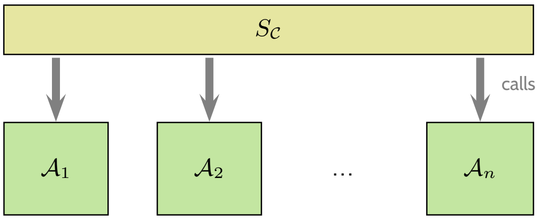
- (Jacobson et al., 1992): "System development is model building."
- Model **driven** software engineering (MDSE): **everything** is a model.
- Model **based** software engineering (MBSE): **some** models are used.



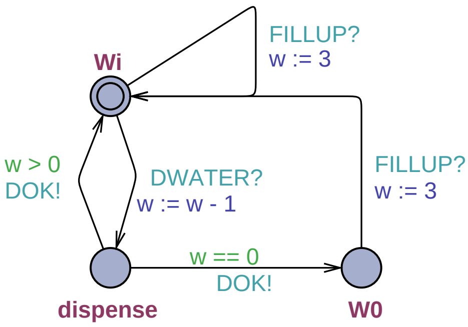
Implementing CFA

- Now that we have a CFA **model** $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ (**thoroughly checked** using Uppaal), we would like to have **software** – an implementation of the model.
- This task can be split into two sub-tasks:
  - (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**.)



Example



st : { Wi, W0, dispense }
w : int = 3

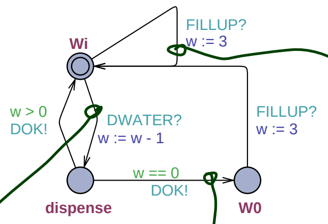
⋮
st = Wi : if (α = DWATER?)
w := w - 1; st := dispense;
else (α = FILLUP?)
w := 3;

Example

```
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 :     st := W0;
                                                 R := R ∪ {FILLUP?};
                        □ α = DOK! ∧ w > 0 :     st := Wi;
                                                 R := R ∪ {FILLUP?};
                        fi;
    □ st = W0 :         if
                        □ α = FILLUP? :          w := 3;
                                                 st := Wi;
                                                 R := R ∪ {FILLUP?, DWATER?};
                        fi;
    fi;
    return R;
}
```



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;
}
```

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**.

Putting It All Together

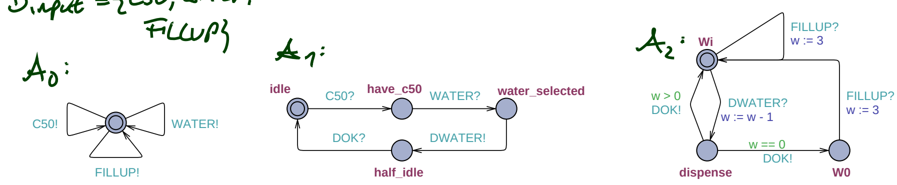
- Let $\mathcal{N} = \mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ with **pairwise disjoint variables**.
- Assume $B = B_{input} \dot\cup B_{internal}$, where B_{input} are **dedicated input channels**, i.e. there is no edge with action $a!$ and $a \in B_{input}$.
- Then software $S_\mathcal{N}$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

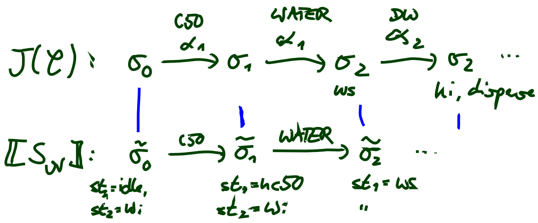
```
Set⟨Act⟩ R_1 := R_{1,ini}, ..., R_n := R_{n,ini} ;   // initially enabled actions
void main() {
  do
  □ true : if
           □ true : (α, snd, rcv) := select(R_1, ..., R_n);   // choose synchronisation
                                                             //   (rcv = 0 if α = τ,
                                                             //   blocks on deadlock)
           □ true : (α, snd, rcv) := read_input();           // or read input (snd = 0)
           fi
           for (k = 1 to n) if (snd = k) R_k := take_action_k(α);   // sender
           for (k = 1 to n) if (rcv = k) R_k := take_action_k(ᾱ);   // receiver
           // snapshot
  od
}
```

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** $\mathcal{T}(\mathcal{C}(\mathcal{A}_0, \mathcal{A}_1, \dots, \mathcal{A}_n))$ where $\mathcal{A}_0$ has one location ℓ and edges

$$E_0 = \{(\ell, \alpha!, \textit{true}, \langle\rangle, \ell) \mid \alpha \in B_{input}\}.$$

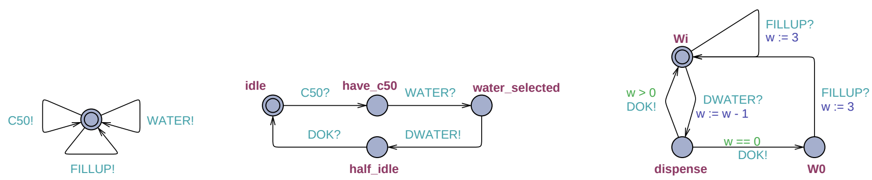




Model vs. Implementation

- 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** $\mathcal{T}(\mathcal{C}(\mathcal{A}_0, \mathcal{A}_1, \dots, \mathcal{A}_n))$ where $\mathcal{A}_0$ has one location ℓ and edges

$$E_0 = \{(\ell, \alpha!, \textit{true}, \langle\rangle, \ell) \mid \alpha \in B_{input}\}.$$



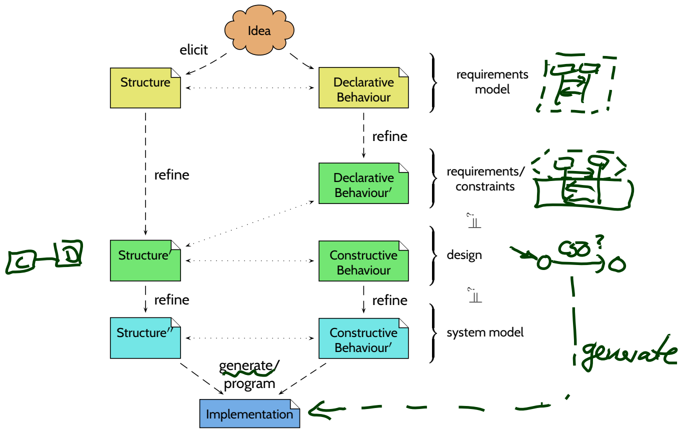
- Yes, and...?
  - If Uppaal reports that $\mathcal{N}_{\text{VM}} \models \exists\Diamond\, w = 0$ holds, then $w = 0$ **is reachable** in $[\![S_{\mathcal{N}_{\text{VM}}}]\!]$.
  - If Uppaal reports that

    $$\mathcal{N}_{\text{VM}} \models \forall\Box\, \texttt{tea\_enabled} \quad \texttt{imply} \quad \text{CoinValidator}.\texttt{have\_c150}$$

    holds, then $[\![S_{\mathcal{N}_{\text{VM}}}]\!]$ **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.

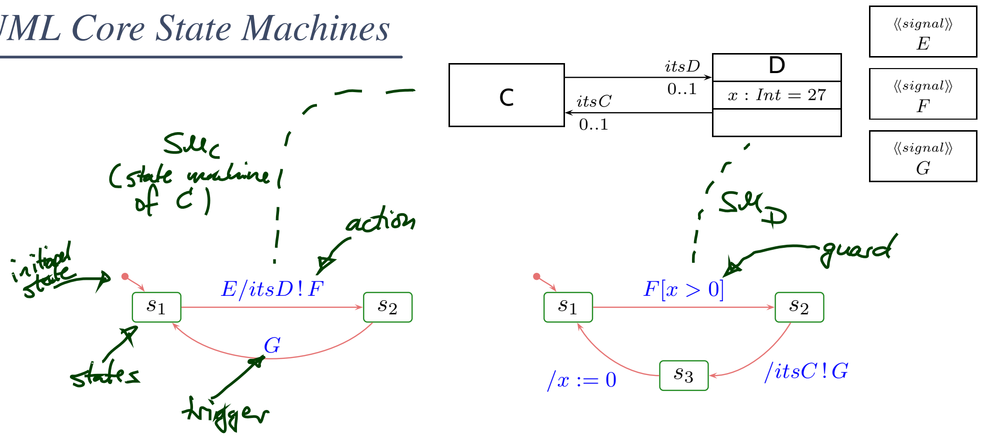


Content

- **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

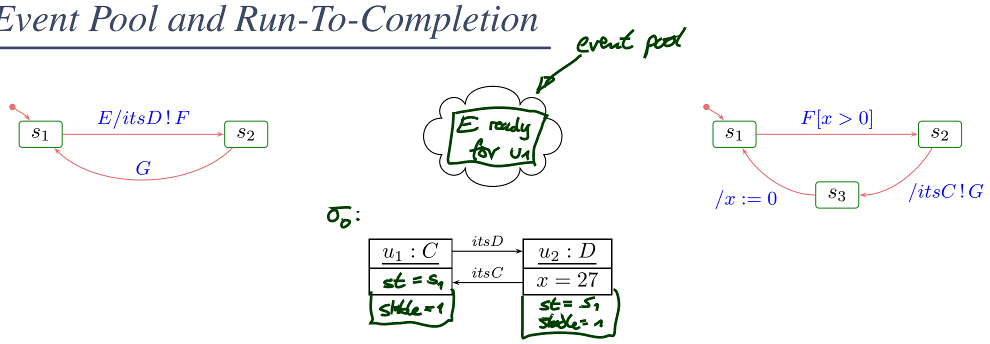


$$annot ::= \big[\underbrace{\langle event\rangle [\,\texttt{.}\,\langle event\rangle]^*}_{trigger} \quad [\,\texttt{[}\,\langle guard\rangle\,\texttt{]}\,] \quad [\,\texttt{/}\,\langle action\rangle]\,\big]$$

with

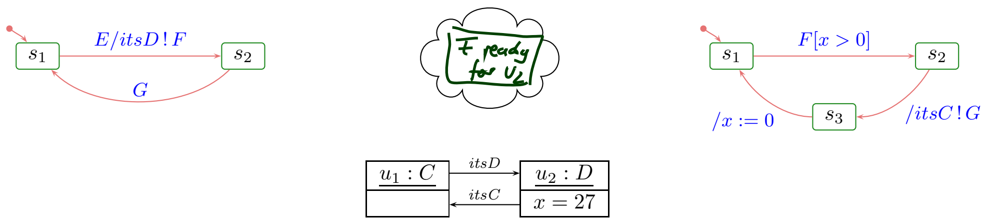
- $event \in \mathcal{E}$, (optional)
- $guard \in Expr_{\mathscr{S}}$ (default: *true*, assumed to be in $Expr_{\mathscr{S}}$)
- $action \in Act_{\mathscr{S}}$ (default: `skip`, assumed to be in $Act_{\mathscr{S}}$)

Event Pool and Run-To-Completion



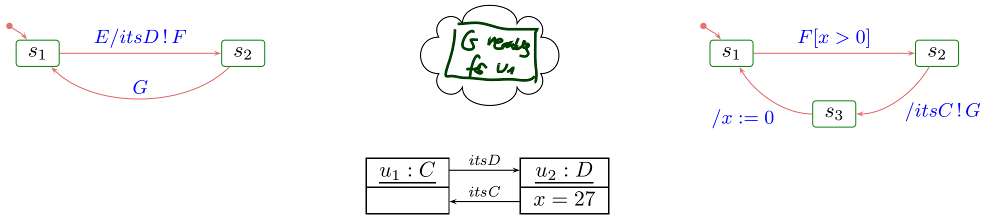
step	u_1		u_2			event pool
	state	stable	x	state	stable	
0	s_1	1	27	s_1	1	E ready for u_1

Event Pool and Run-To-Completion



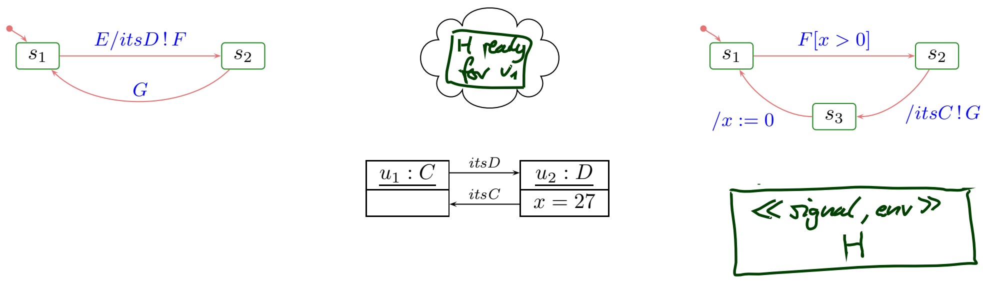
step	u_1		u_2			event pool
	state	stable	x	state	stable	
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2

Event Pool and Run-To-Completion



step	u_1 state	u_1 stable	x	u_2 state	u_2 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

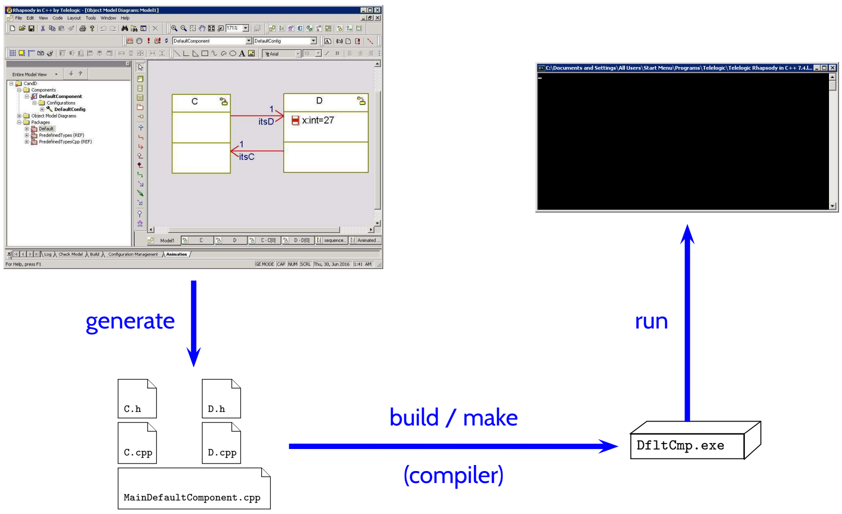


step	u_1 state	u_1 stable	x	u_2 state	u_2 stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1
4.a	s_2	1	0	s_1	1	G ready for u_1
5.a	s_1	1	0	s_1	1	
4.b	s_1	1	27	s_3	0	
5.b	s_1	1	0	s_1	1	
6	—"—	—	—"—	—		H ready for u_1

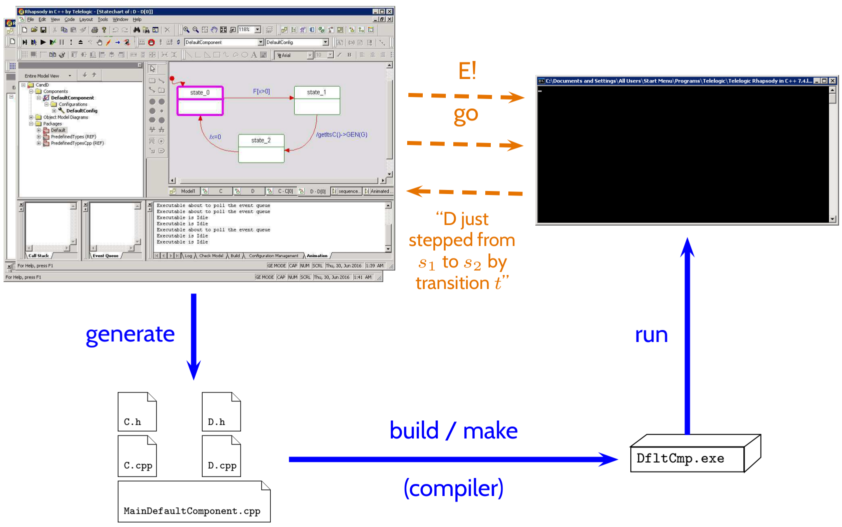
same (5.a ↔ 5.b)

↳ discard H

Rhapsody Architecture

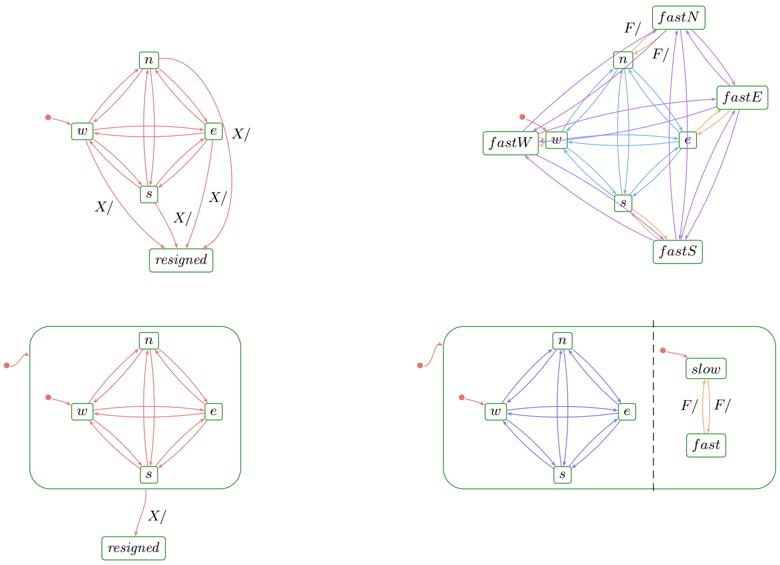


Rhapsody Architecture

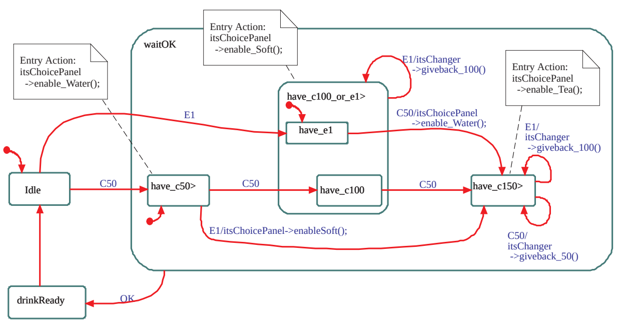


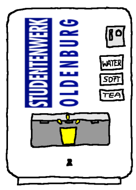
Composite (or Hierarchical) States

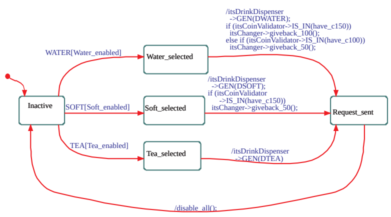
- OR-states, AND-states Harel (1987).
- Composite states are about **abbreviation**, **structuring**, and **avoiding redundancy**.

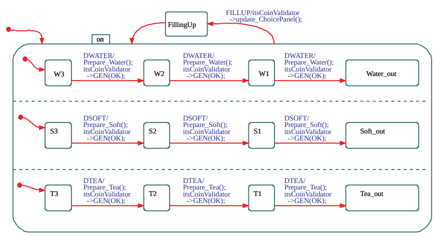


Example

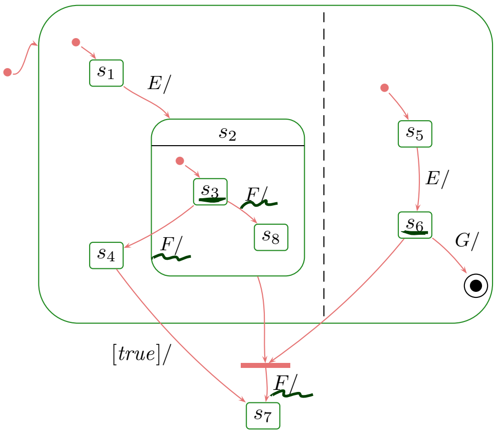








Would be Too Easy...



→ "**Software Design, Modelling, and Analysis with UML**" in the winter semester.

UML Modes

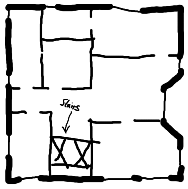
UML and the Pragmatic Attribute

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

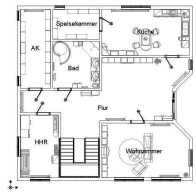
(iii) the **pragmatic attribute**,
i.e. the model is built in a specific context for a specific purpose.

Examples for context/purpose:

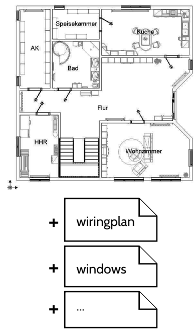
Floorplan as sketch:



Floorplan as blueprint:



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	**Blueprint**	**ProgrammingLanguage**
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"*	*[...] 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. [...]*	*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**.

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.

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.

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.