Softwaretechnik / Software-Engineering

Lecture 13: Behavioural Software Modelling

2017-07-06

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

VL 14

    - b) Uppaal query language
    - c) basic state-machines
    - d) an outlook on hierarchical state-machines

⋮

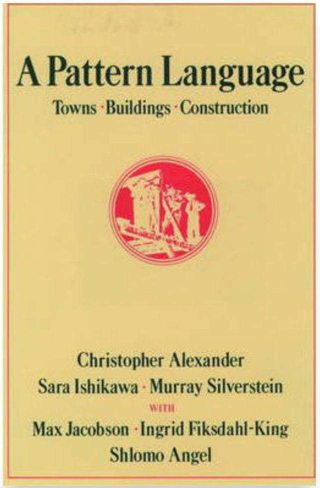
Content

- **Design Patterns**
  - Strategy, Examples
- **Communicating Finite Automata** (CFA)
  - concrete and abstract syntax,
  - networks of CFA,
  - operational semantics.
- **Transition Sequences**
- **Deadlock, Reachability**
- **Uppaal**
  - tool demo (simulator),
  - query language,
  - CFA model-checking.
- **CFA at Work**
  - drive to configuration, scenarios, invariants
  - tool demo (verifier).
- **CFA vs. Software**

Design Patterns

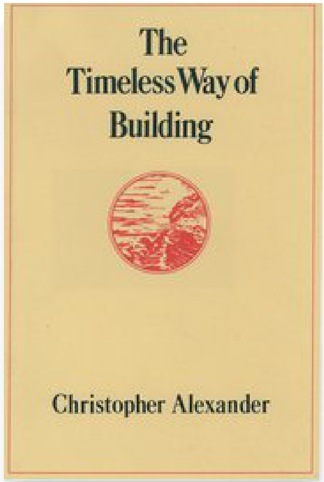
Design Patterns

- In a sense the same as **architectural patterns**, but on a lower scale.
- Often traced back to (Alexander et al., 1977; Alexander, 1979).



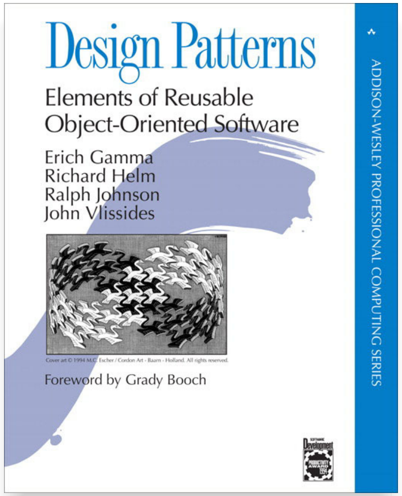
A Pattern Language
Towns · Buildings · Construction

Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel



The
Timeless Way of
Building

Christopher Alexander



Design Patterns
Elements of Reusable
Object-Oriented Software

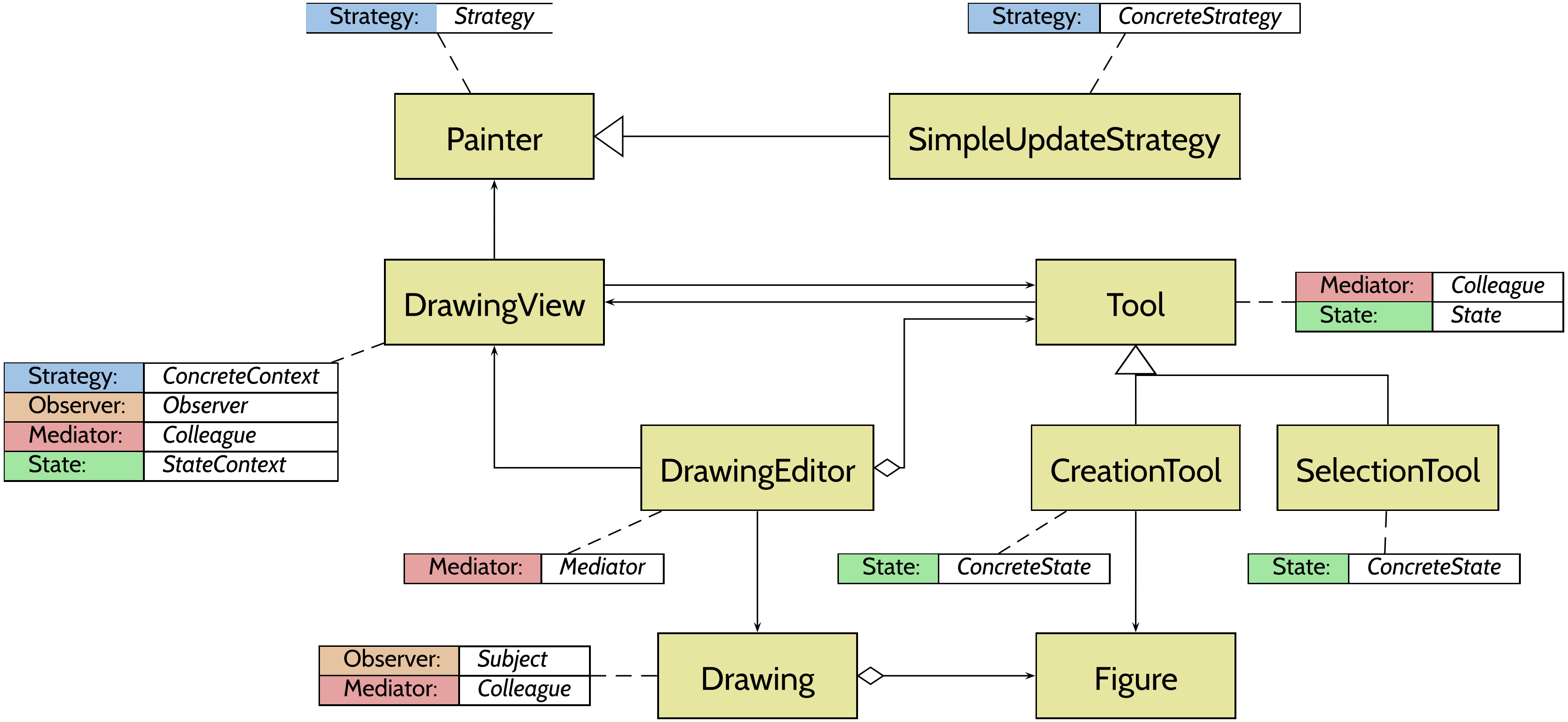
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

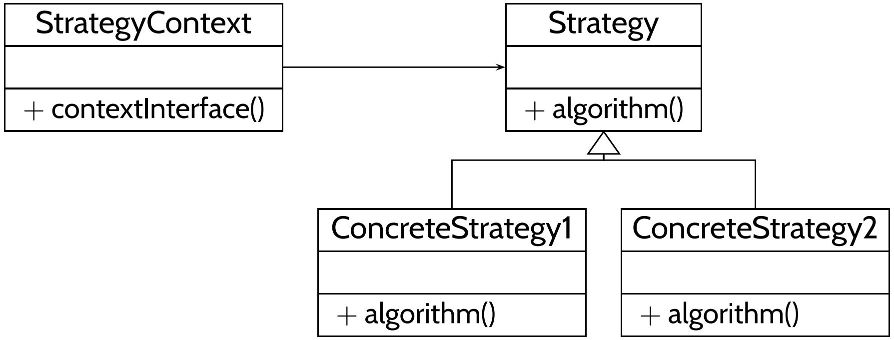
> **Design patterns** ... are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
>
> A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.
>
> (Gamma et al., 1995)

Example: Pattern Usage and Documentation

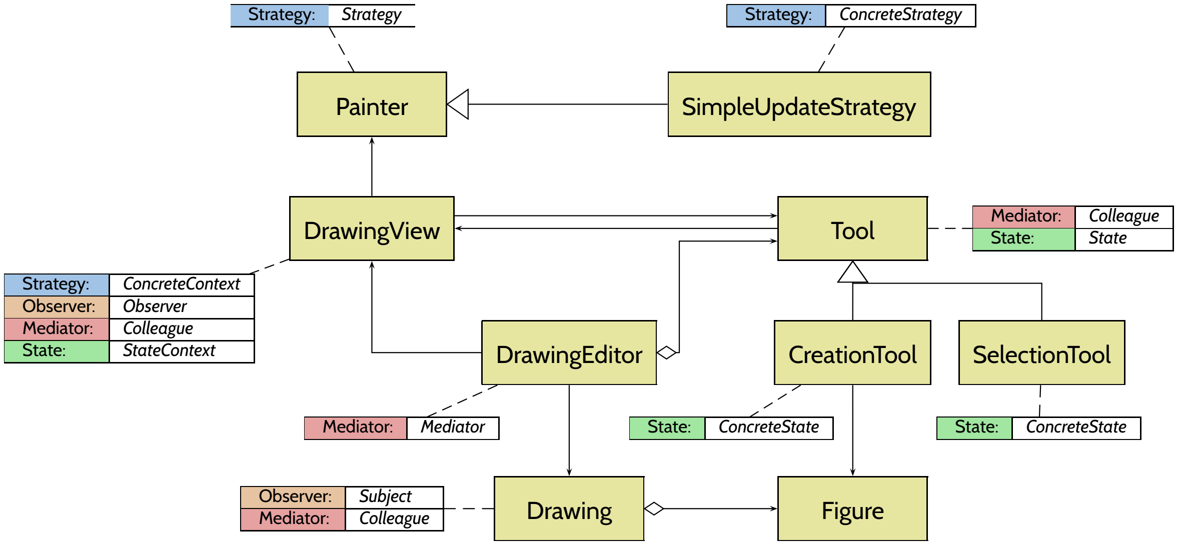


Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

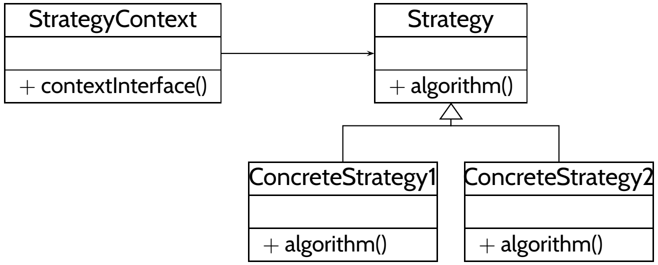
Example: Strategy

	Strategy
Problem	The only difference between similar classes is that they solve the same problem by different algorithms.
Solution	- Have one class **StrategyContext** with all common operations. - Another class **Strategy** provides signatures for all operations to be implemented differently. - From Strategy, derive one sub-class **ConcreteStrategy** for each implementation alternative. - StrategyContext uses concrete Strategy-objects to execute the different implementations via delegation.
Structure	

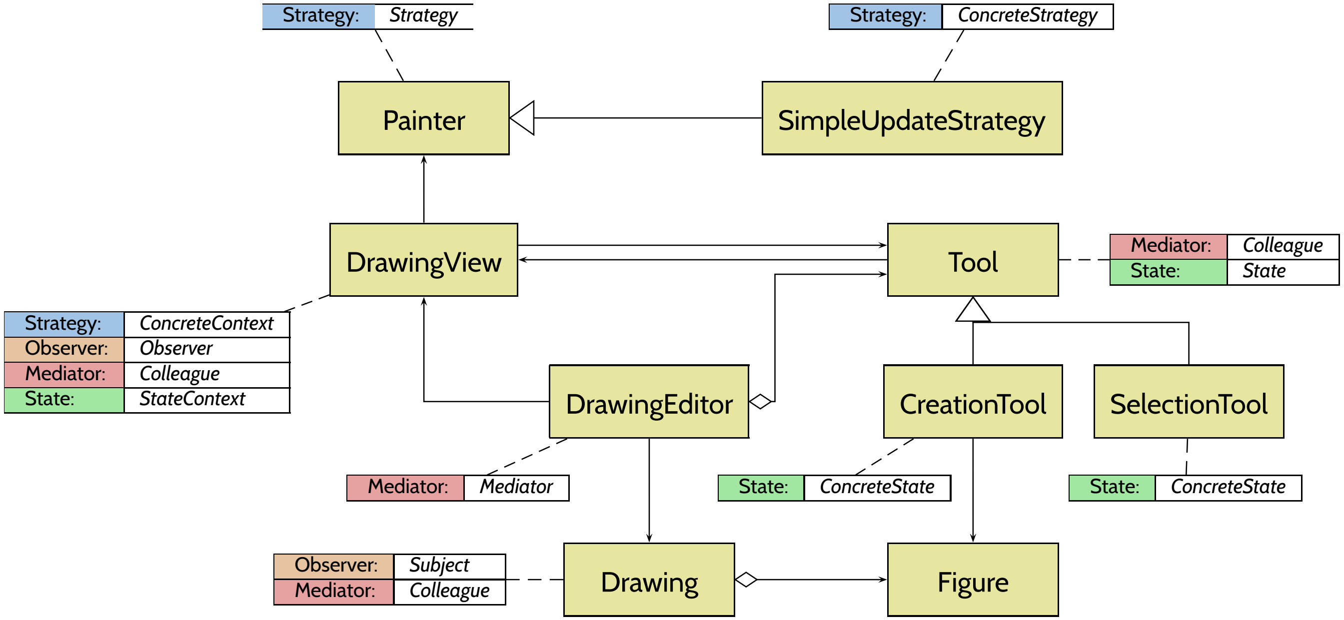
Example: Pattern Usage and Documentation



Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

	Strategy
Problem	The only difference between similar classes is that they solve the same problem by different algorithms.
Solution	…
Structure	

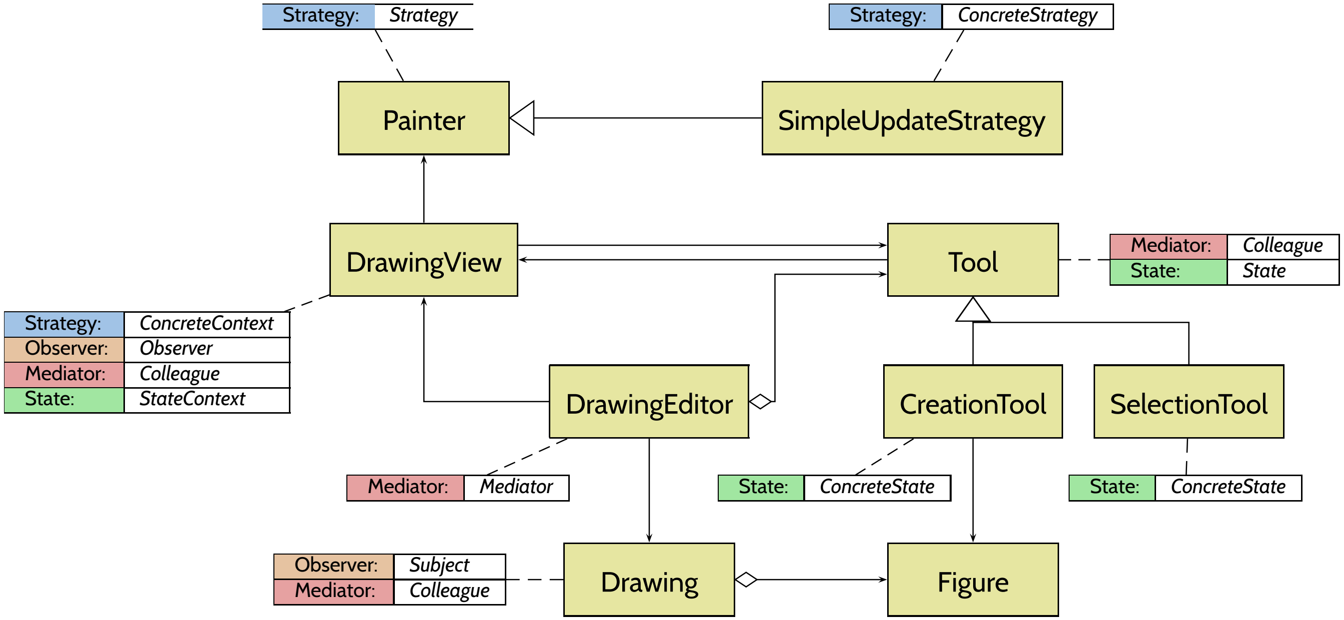
Example: Pattern Usage and Documentation



Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

	Observer
Problem	Multiple objects need to adjust their state if one particular other object is changed.
Example	All GUI object displaying a file system need to change if files are added or removed.

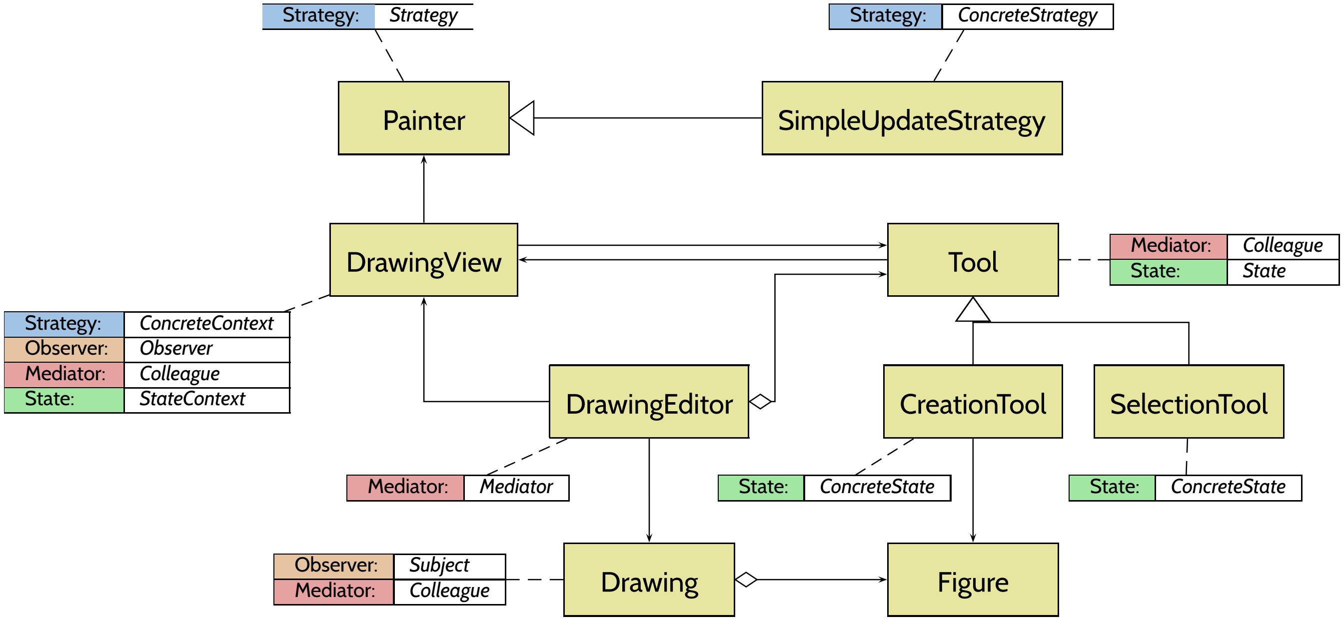
Example: Pattern Usage and Documentation



Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

	State
Problem	The behaviour of an object depends on its (internal) state.
Example	The effect of pressing the room ventilation button depends (among others?) on whether the ventilation is on or off.

Example: Pattern Usage and Documentation



Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

	Mediator
Problem	Objects interacting in a complex way should only be loosely coupled and be easily exchangeable.
Example	Appearance and state of different means of interaction (menus, buttons, input fields) in a graphical user interface (GUI) should be consistent in each interaction state.

Other Patterns: Singleton and Memento

	Singleton
Problem	Of one class, exactly one instance should exist in the system.
Example	Print spooler.

	Memento
Problem	The state of an object needs to be archived in a way that allows to re-construct this state without violating the principle of data encapsulation.
Example	Undo mechanism.

Design Patterns: Discussion

> “The development of design patterns is considered to be one of the most important innovations of software engineering in recent years.”
>
> (Ludewig and Lichter, 2013)

- **Advantages**:
  - **(Re-)use** the experience of others and employ well-proven solutions.
  - Can improve on **quality criteria** like changeability or re-use.
  - Provide a **vocabulary** for the design process,
    thus facilitates documentation of architectures and discussions about architecture.
  - Can be combined in a flexible way,
    one class in a particular architecture can correspond to roles of multiple patterns.
  - Helps teaching software design.

- **Disadvantages**:
  - Using a pattern is not a value as such.
    Having too much global data cannot be justified by “but it’s the pattern Singleton”.
  - **Again**: reading is easy, writing need not be.

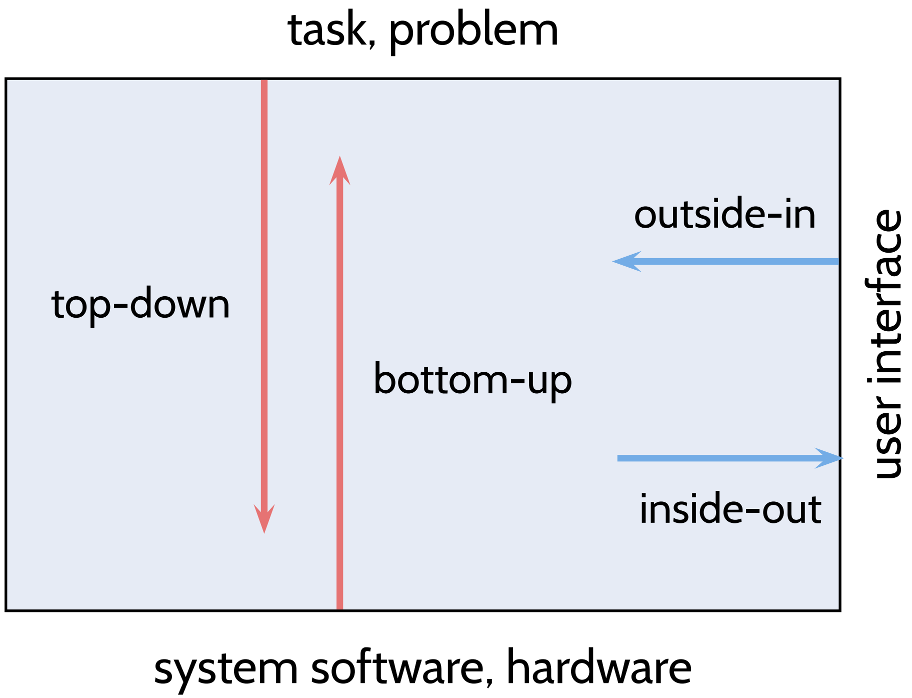
    Here: Understanding abstract descriptions of design patterns or their use in existing software may be easy – using design patterns appropriately in new designs requires (**surprise, surprise**) experience.

Quality Criteria on Architectures

Quality Criteria on Architectures

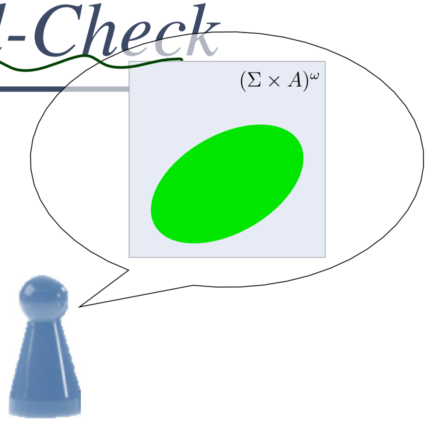
- **testability**
  - architecture design should keep testing (or formal verification) in mind (**buzzword** "design for verification"),
  - high locality of design units may make testing significantly easier (module testing),
  - particular testing interfaces may improve testability (e.g. allow injection of user input not only via GUI; or provide particular log output for tests).
- **changeability**, **maintainability**
  - most systems that are used need to be changed or maintained, in particular when requirements change,
  - **risk assessment**: parts of the system with high probability for changes should be designed such that changes are possible with acceptable effort (abstract, modularise, encapsulate),
- **portability**
  - **porting**: adaptation to different platform (OS, hardware, infrastructure).
  - systems with a long lifetime may need to be adapted to different platforms over time, infrastructure like databases may change (→ introduce abstraction layer).

Development Approaches

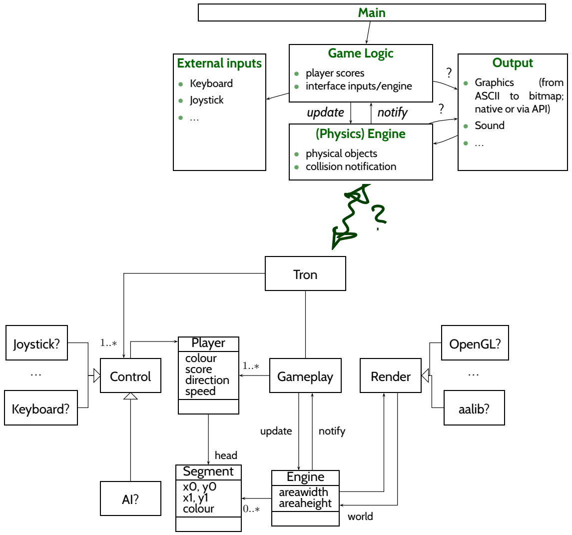


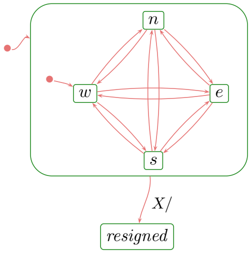
- **top-down** risk: needed functionality hard to realise on target platform.
- **bottom-up** risk: lower-level units do not “fit together”.
- **inside-out** risk: user interface needed by customer hard to realise with existing system,
- **outside-in** risk: elegant system design not reflected nicely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

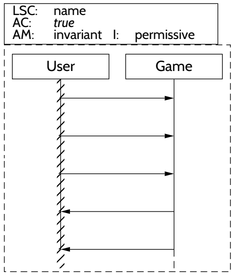
Transform vs. Write-Down-and-Check



Analyst



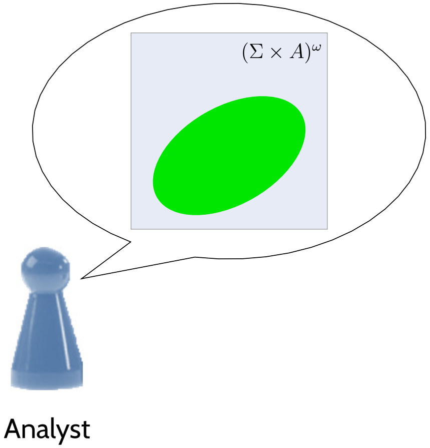


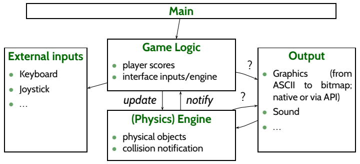


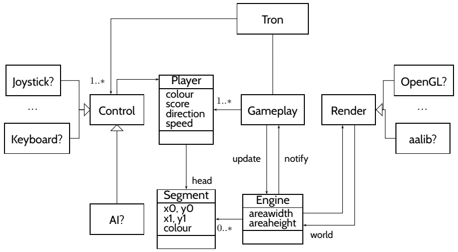
Content

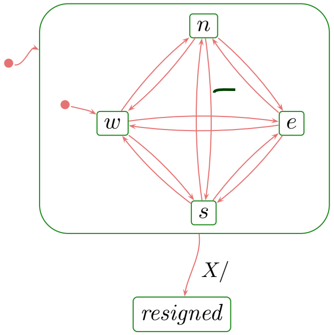
- **Design Patterns**
  - Strategy, Examples
- **Communicating Finite Automata** (CFA)
  - concrete and abstract syntax,
  - networks of CFA,
  - operational semantics.
- **Transition Sequences**
- **Deadlock, Reachability**
- **Uppaal**
  - tool demo (simulator),
  - query language,
  - CFA model-checking.
- **CFA at Work**
  - drive to configuration, scenarios, invariants
  - tool demo (verifier).
- **CFA vs. Software**

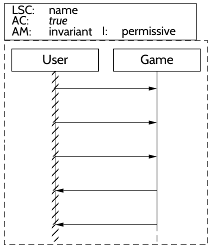
Software Modelling







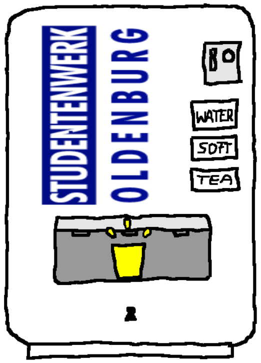




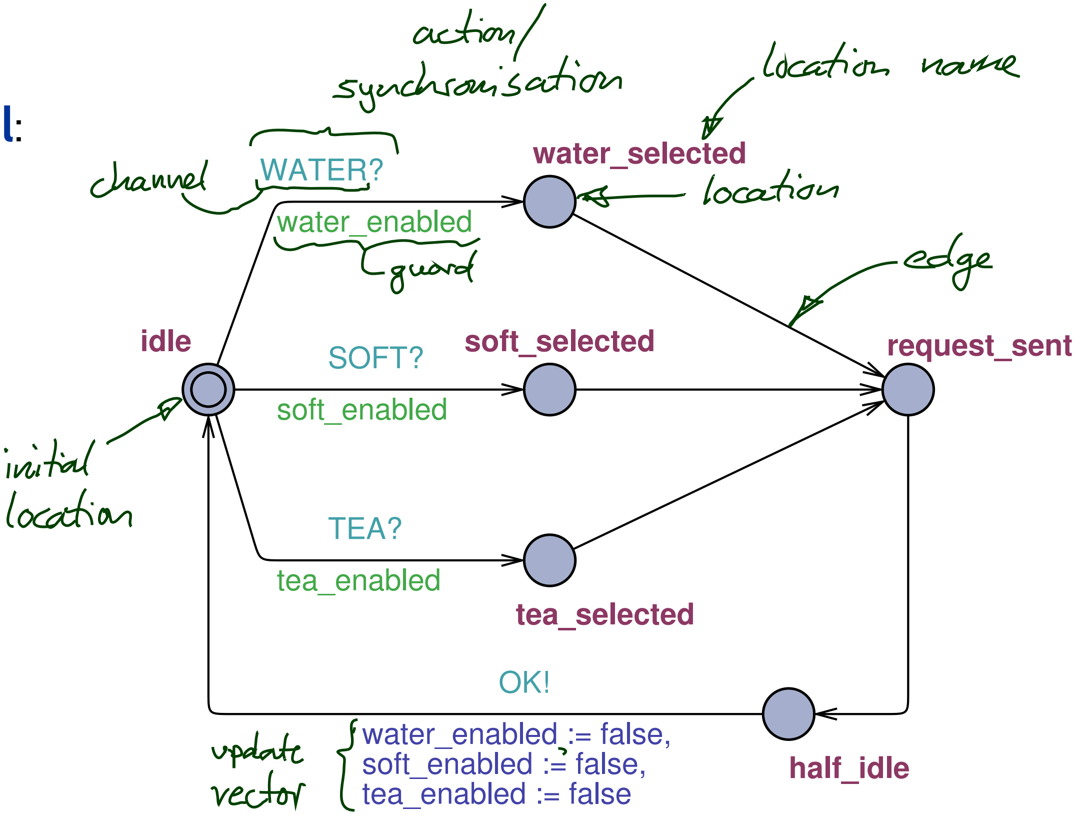
Communicating Finite Automata

presentation follows (Olderog and Dierks, 2008)

Example



ChoicePanel:
(simplified)



Channel Names and Actions

To define communicating finite automata, we need the following sets of symbols:

- A set $(a, b \in)$ $\undertilde{\mathsf{Chan}}$ of **channel names** or **channels**.
- For each channel $a \in \mathsf{Chan}$, two **visible actions**:
  $a?$ and $a!$ denote **input** and **output** on the **channel** $(a?, a! \notin \mathsf{Chan})$.
- $\tau \notin \mathsf{Chan}$ represents an **internal action**, not visible from outside.
- $(\alpha, \beta \in)$ $Act := \{a? \mid a \in \mathsf{Chan}\} \cup \{a! \mid a \in \mathsf{Chan}\} \cup \{\tau\}$ is the set of **actions**.

- An **alphabet** B is a set of **channels**, i.e. $B \subseteq \mathsf{Chan}$.
- For each alphabet B, we define the corresponding **action set**

$$B_{?!} := \{a? \mid a \in B\} \cup \{a! \mid a \in B\} \cup \{\tau\}.$$

Note: $\mathsf{Chan}_{?!} = Act$.

Integer Variables and Expressions, Resets

- Let $(v, w \in)\ V$ be a set of ((**finite domain**) integer) variables.

  By $(\varphi \in)\ \Psi(V)$ we denote the set of **integer expressions** over V using function symbols $+, -, \ldots$ and relation symbols $<, \leq, \ldots$.

- A **modification** on $\overset{\in V}{v}$ is of the form

  $$v := \varphi, \qquad v \in V, \quad \varphi \in \Psi(V).$$

  By $R(V)$ we denote the set of all modifications.

- By $\vec{r}$ we denote a finite list $\langle r_1, \ldots, r_n \rangle$, $n \in \mathbb{N}_0$, of modifications $r_i \in R(V)$. $\vec{r}$ is called **reset vector** (or **update vector**).

  $\langle\rangle$ is the empty list ($n = 0$).

- By $R(V)^*$ we denote the set of all such finite lists of modifications.

Communicating Finite Automata

Definition. A **communicating finite automaton** is a structure

$$\mathcal{A} = (L, B, V, E, \ell_{ini})$$

where

- ($\ell \in$) L is a finite set of **locations** (or **control states**),
- $B \subseteq$ Chan,
- V: a set of data variables,
- $E \subseteq L \times B_{!?} \times \Phi(V) \times R(V)^* \times L$: a finite set of **directed edges** such that

  $$\underline{(\ell, \alpha, \varphi, \vec{r}, \ell')} \in E \wedge \mathsf{chan}(\alpha) \in U \implies \varphi = \mathit{true}.$$

  Edges $(\ell, \alpha, \varphi, \vec{r}, \ell')$ from location ℓ to ℓ' are labelled with an **action** α, a **guard** φ, and a list $\vec{r}$ of **modifications**.
- ℓ_{ini} is the **initial location**.

  $\in L$

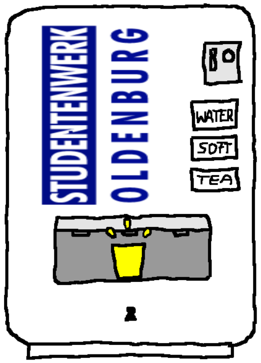
Example

$L = \{ idle, water\_selected, \dots \}$

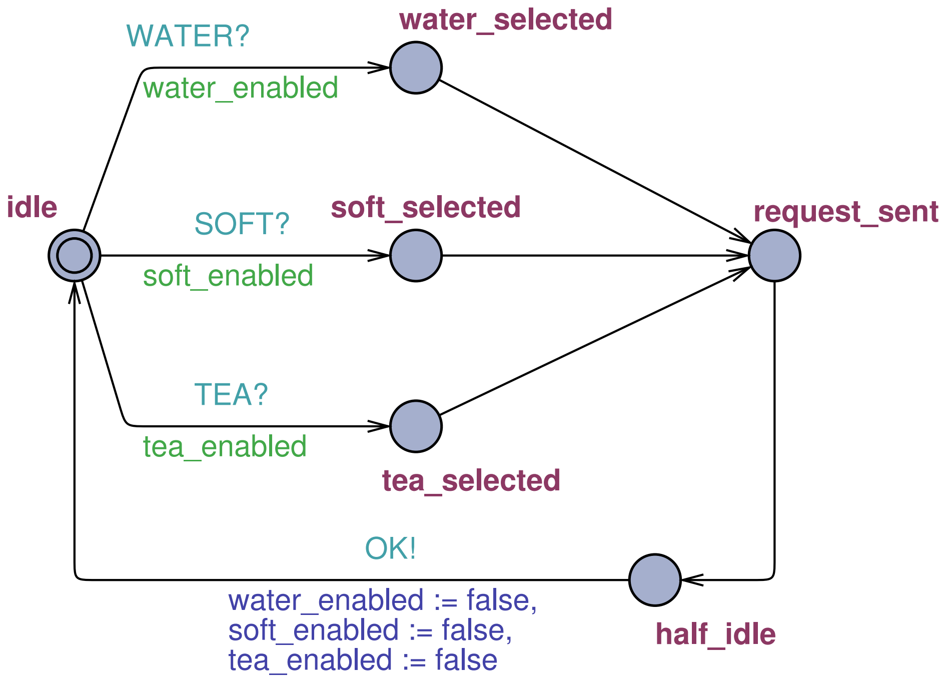
$B = \{ WATER, SOFT, OK, \dots \}$

$V = \{ water\_enabled, \dots \}$

$\ell_{ini} = idle$



ChoicePanel:
(simplified)



$$E = \{ (\underset{\underset{L}{\cap}}{idle}, \underset{\underset{B_{?!}}{\cap}}{WATER?}, \underset{\underset{\Phi(V)}{\cap}}{water\_enabled}, \underset{\underset{R(V)^*}{\cap}}{\langle\rangle}, water\_\underset{\underset{L}{\cap}}{\overset{selected}{\cancel{enabled}}}), \dots \}$$

Operational Semantics of Networks of CFA

Definition.
Let $\mathcal{A}_i = (L_i, B_i, V_i, E_i, \ell_{ini,i})$, $1 \leq i \leq n$, be communicating finite automata.

The **operational semantics** of the **network** of CFA $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$
is the labelled transition system

$$\mathcal{T}(\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)) = (\mathit{Conf}, \underbrace{\mathsf{Chan} \cup \{\tau\}}_{\text{labels}}, \{\xrightarrow{\lambda} \mid \lambda \in \underline{\mathsf{Chan} \cup \{\tau\}}\}, C_{ini})$$

where

$= (\ell_1, \dots, \ell_n)$ location vector

valuation

- $V = \bigcup_{i=1}^{n} V_i$,
- $\mathit{Conf} = \{\langle \vec{\ell}, \nu \rangle \mid \ell_i \in L_i, \nu : V \to \mathscr{D}(V)\}$,
- $C_{ini} = \langle \vec{\ell}_{ini}, \nu_{ini} \rangle$ with $\nu_{ini}(v) = 0$ for all $v \in V$.

The transition relation consists of transitions of the following two types.

Helpers: Extended Valuations and Effect of Resets

- $\nu : V \to \mathscr{D}(V)$ is a **valuation** of the variables,
- A valuation ν of the variables canonically assigns an integer value $\nu(\varphi)$ to each integer expression $\varphi \in \Phi(V)$.
- $\models \subseteq (V \to \mathscr{D}(V)) \times \Phi(V)$ is the canonical **satisfaction relation** between valuations and integer expressions from $\Phi(V)$.

- **Effect of modification** $r \in R(V)$ **on** ν, denoted by $\nu[r]$:

$$\underbrace{(\nu[\underbrace{v := \varphi}_{}])}_{: V \to \mathscr{D}(V)}(a) := \begin{cases} \nu(\varphi), \text{ if } a = v, \\ \nu(a), \text{ otherwise} \end{cases}$$

- We set $\nu[\langle r_1, \ldots, r_n \rangle] := \nu[r_1] \ldots [r_n] = (((\nu[r_1])[r_2]) \ldots)[r_n]$.

  That is, modifications are executed sequentially from left to right.

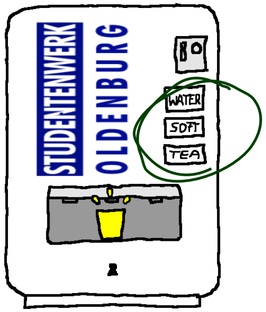
Operational Semantics of Networks of CFA

- An **internal transition** $\langle \vec{\ell}, \nu \rangle \xrightarrow{\tau} \langle \vec{\ell}', \nu' \rangle$ occurs if there is $i \in \{1, \dots, n\}$ and
  - there is a τ-edge $(\ell_i, \tau, \varphi, \vec{r}, \ell_i') \in E_i$ such that
    - $\nu \models \varphi$, "source valuation satisfies guard"
    - $\vec{\ell}' = \vec{\ell}[\ell_i := \ell_i']$, "automaton i changes location"
    - $\nu' = \nu[\vec{r}]$, "ν' is the result of applying $\vec{r}$ on ν"

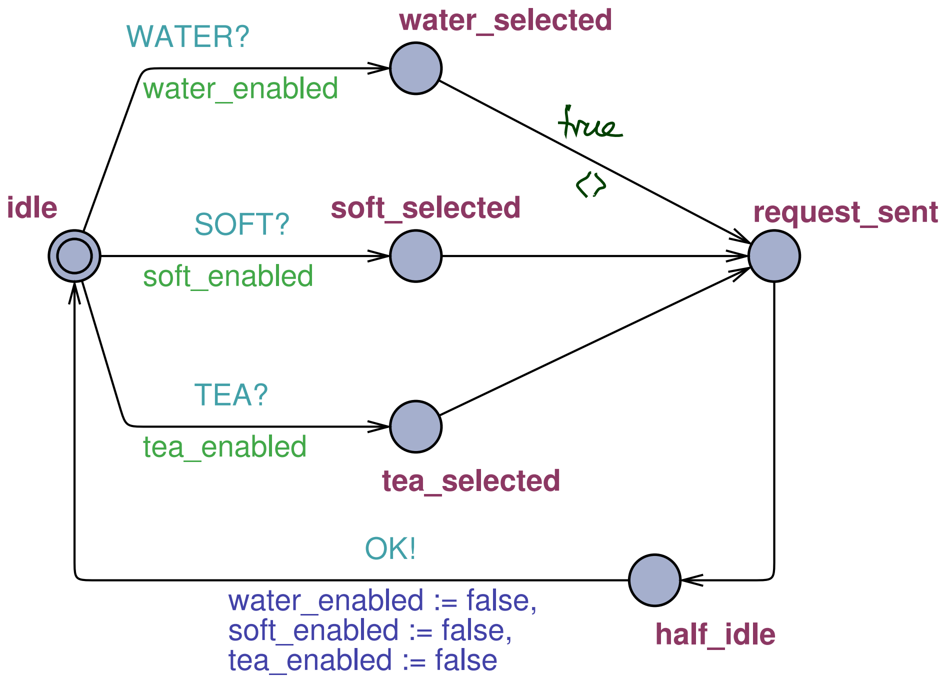
- A **synchronisation transition** $\langle \vec{\ell}, \nu \rangle \xrightarrow{b} \langle \vec{\ell}', \nu' \rangle$ occurs if there are $i, j \in \{1, \dots, n\}$ with $i \neq j$ and
  - there are edges $(\ell_i, \underline{b!}, \varphi_i, \vec{r}_i, \ell_i') \in E_i$ and $(\ell_j, \underline{b?}, \varphi_j, \vec{r}_j, \ell_j') \in E_j$ such that
    - $\nu \models \varphi_i \wedge \varphi_j$, "source valuation satisfies guards (!)"
    - $\vec{\ell}' = \vec{\ell}[\ell_i := \ell_i'][\ell_j := \ell_j']$, "automaton i and j change location"
    - $\nu' = (\nu[\vec{r}_i])[\vec{r}_j]$, "$\nu'$ is the result of applying first $\vec{r}_i$ and then $\vec{r}_j$ on ν"

This style of communication is known under the names "**rendezvous**", "**synchronous**", "**blocking**" communication (and possibly many others).

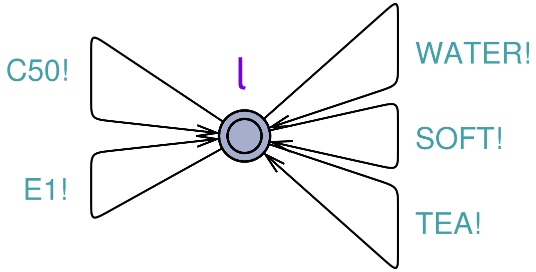
Example



ChoicePanel:
(simplified)



User:



$$\underbrace{\left\langle (idle, \ell), \begin{smallmatrix} we=1 \\ se=1 \\ te=0 \end{smallmatrix} \right\rangle \xrightarrow{WATER} \left\langle (water\_selected, \ell), \begin{smallmatrix} we=1 \\ se=1 \\ te=0 \end{smallmatrix} \right\rangle} \xrightarrow{\tau} \underbrace{\left\langle (request\_sent, \ell), \begin{smallmatrix} we=1 \\ se=1 \\ te=0 \end{smallmatrix} \right\rangle}$$

Transition Sequences

- A **transition sequence** of $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ is any (in)finite sequence of the form

$$\underbrace{\underline{\langle \vec{\ell}_0, \nu_0 \rangle} \xrightarrow{\lambda_1} \rlap{$\overbrace{\phantom{\langle \vec{\ell}_1, \nu_1 \rangle \xrightarrow{\lambda_2} \langle \vec{\ell}_2, \nu_2 \rangle}}$}\langle \vec{\ell}_1, \nu_1 \rangle}_{} \xrightarrow{\lambda_2} \underbrace{\langle \vec{\ell}_2, \nu_2 \rangle \xrightarrow{\lambda_3} \dots}_{}$$

  with

  - $\langle \vec{\ell}_0, \nu_0 \rangle = C_{ini}$,
  - for all $i \in \mathbb{N}$, there is $\xrightarrow{\lambda_{i+1}}$ in $\mathcal{T}(\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n))$ with $\langle \vec{\ell}_i, \nu_i \rangle \xrightarrow{\lambda_{i+1}} \langle \vec{\ell}_{i+1}, \nu_{i+1} \rangle$.

Deadlock

- A **configuration** $\langle \vec{\ell}, \nu \rangle$ of $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ is called **deadlock** if and only if there are no transitions from $\langle \ell, \nu \rangle$, i.e. if

$$\neg(\exists\, \lambda \in \Lambda\, \exists\, \langle \vec{\ell}', \nu' \rangle \in \mathit{Conf} \bullet \langle \vec{\ell}, \nu \rangle \xrightarrow{\lambda} \langle \vec{\ell}', \nu' \rangle).$$

  The **network** $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ is said to **have a deadlock** if and only if there is a reachable configuration $\langle \vec{\ell}, \nu \rangle$ which is a deadlock.

Reachability

- A **configuration** $\langle \vec{\ell}, \nu \rangle$ is called **reachable** (in $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$) **from** $\langle \vec{\ell}_0, \nu_0 \rangle$ if and only if there is a transition sequence of the form
$$\langle \vec{\ell}_0, \nu_0 \rangle \xrightarrow{\lambda_1} \langle \vec{\ell}_1, \nu_1 \rangle \xrightarrow{\lambda_2} \langle \vec{\ell}_2, \nu_2 \rangle \xrightarrow{\lambda_3} \dots \xrightarrow{\lambda_n} \langle \vec{\ell}_n, \nu_n \rangle = \langle \vec{\ell}, \nu \rangle.$$

- A **configuration** $\langle \vec{\ell}, \nu \rangle$ is called **reachable** (without "from"!) if and only if it is reachable from $\underset{\sim\sim\sim}{C_{ini}}$.

- A **location** $\ell \in L_i$ is called **reachable** if and only if **any** configuration $\langle \vec{\ell}, \nu \rangle$ with $\ell_i = \ell$ is reachable, i.e. there exist $\vec{\ell}$ and ν such that $\ell_i = \ell$ and $\langle \vec{\ell}, \nu \rangle$ is reachable.

Uppaal

(Larsen et al., 1997; Behrmann et al., 2004)

Tool Demo

The Uppaal Query Language

Consider $\mathcal{N} = \mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ over data variables V.

- **basic formula**:

  $$atom ::= \mathcal{A}_i.\ell \mid \varphi \mid \texttt{deadlock}$$

  where $\ell \in L_i$ is a location and φ an expression over V.

- **configuration formulae**:

  $$term ::= atom \mid \texttt{not}\ term \mid term_1\ \texttt{and}\ term_2$$

- **existential path formulae**:

  $$\begin{aligned} e\text{-}formula ::= {} & \exists\Diamond\ term & \text{(exists finally)} \\ \mid {} & \exists\Box\ term & \text{(exists globally)} \end{aligned}$$

- **universal path formulae**:

  $$\begin{aligned} a\text{-}formula ::= {} & \forall\Diamond\ term & \text{(always finally)} \\ \mid {} & \forall\Box\ term & \text{(always globally)} \\ \mid {} & term_1\ \texttt{->}\ term_2 & \text{(leads to)} \end{aligned}$$

- **formulae** (or **queries**):

  $$F ::= e\text{-}formula \mid a\text{-}formula$$

Satisfaction of Uppaal Queries by Configurations

- The **satisfaction relation**

$$\langle \vec{\ell}, \nu \rangle \models F$$

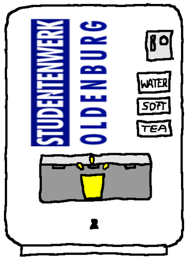
between **configurations**

$$\langle \vec{\ell}, \nu \rangle = \langle (\ell_1, \dots, \ell_n), \nu \rangle$$

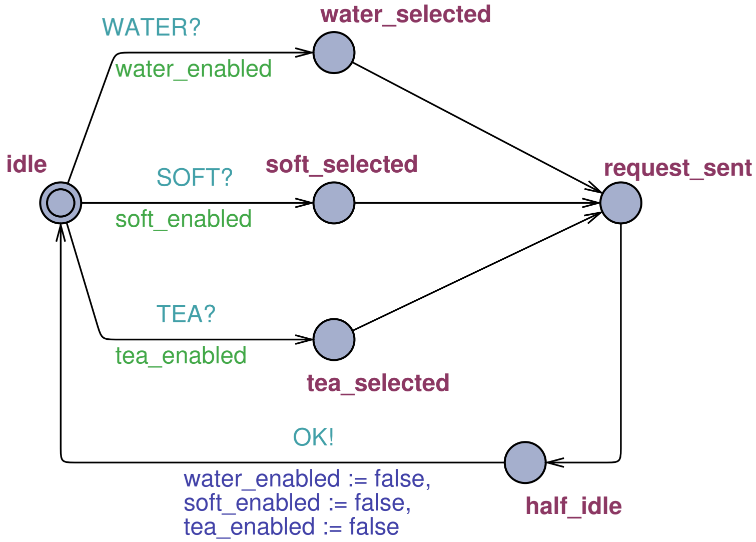
of a network $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ and **formulae** F of the Uppaal logic is defined **inductively** as follows:

- $\langle \vec{\ell}, \nu \rangle \models$ `deadlock` iff $\langle \vec{\ell}, \nu \rangle$ is a deadlock configuration
- $\langle \vec{\ell}, \nu \rangle \models \mathcal{A}_i.\underset{\sim}{\ell}$ iff $\vec{\ell}_i = \underset{\sim}{\ell}$
- $\langle \vec{\ell}, \nu \rangle \models \varphi$ iff $\nu \models \varphi$
- $\langle \vec{\ell}, \nu \rangle \models$ `not` $term$ iff $\langle \vec{\ell}, \nu \rangle \not\models term$
- $\langle \vec{\ell}, \nu \rangle \models term_1$ `and` $term_2$ iff $\langle \vec{\ell}, \nu \rangle \models term_1$ and $\langle \vec{\ell}, \nu \rangle \models term_2$

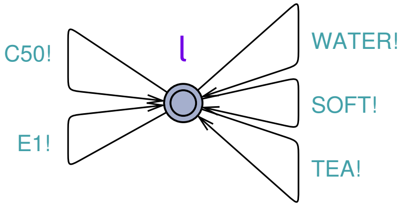
Example: Computation Paths vs. Computation Tree

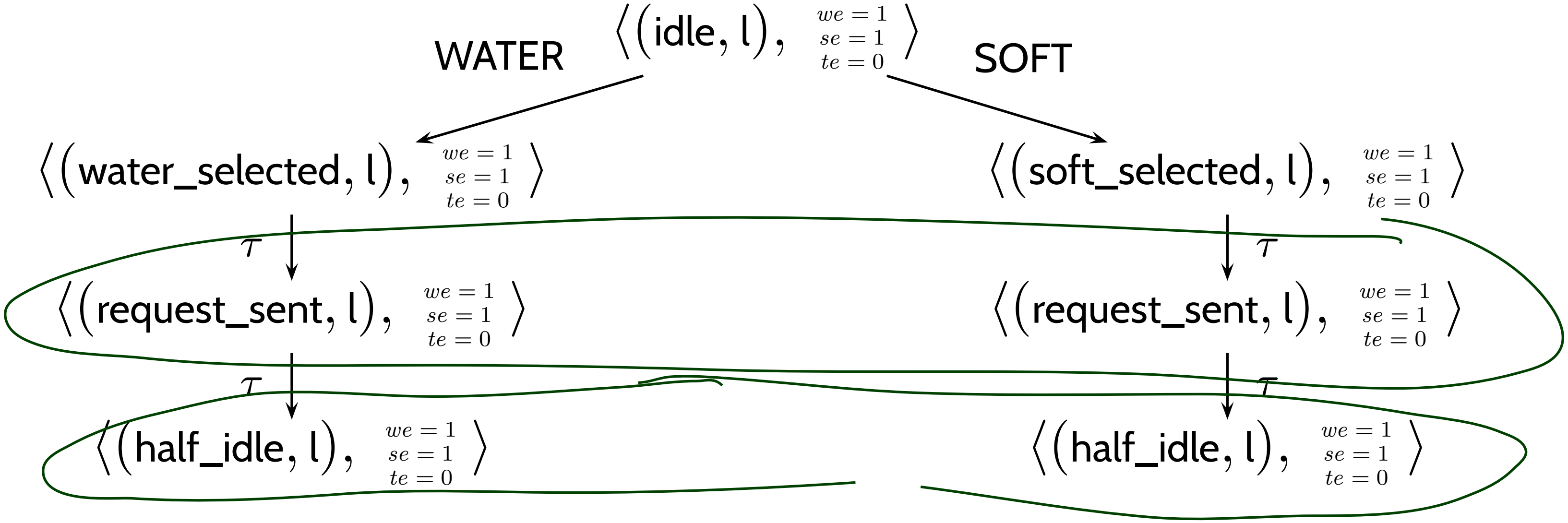


ChoicePanel:



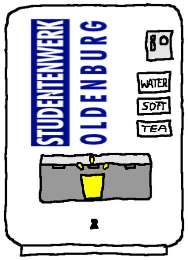
User:



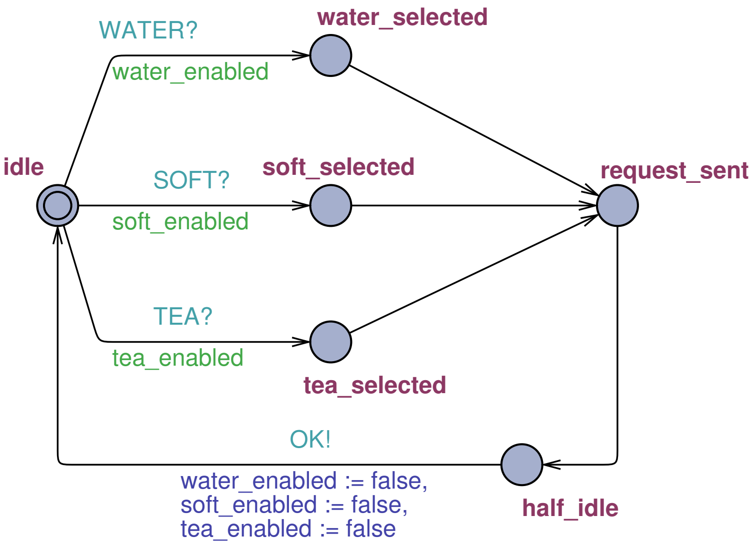


Example: Computation Paths vs. Computation Graph

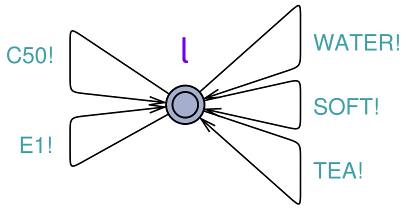
(or: Transition Graph)



ChoicePanel:



User:



$$\begin{tikzcd}
& \left\langle (\mathsf{idle}, \mathsf{l}), \begin{smallmatrix} we = 1 \\ se = 1 \\ te = 0 \end{smallmatrix} \right\rangle \arrow[dl, "\mathsf{WATER}"'] \arrow[dr, "\mathsf{SOFT}"] & \\
\left\langle (\mathsf{water\_selected}, \mathsf{l}), \begin{smallmatrix} we = 1 \\ se = 1 \\ te = 0 \end{smallmatrix} \right\rangle \arrow[dr, "\tau"'] & & \left\langle (\mathsf{soft\_selected}, \mathsf{l}), \begin{smallmatrix} we = 1 \\ se = 1 \\ te = 0 \end{smallmatrix} \right\rangle \arrow[dl, "\tau"] \\
& \left\langle (\mathsf{request\_sent}, \mathsf{l}), \begin{smallmatrix} we = 1 \\ se = 1 \\ te = 0 \end{smallmatrix} \right\rangle \arrow[d, "\tau"'] & \\
& \left\langle (\mathsf{half\_idle}, \mathsf{l}), \begin{smallmatrix} we = 1 \\ se = 1 \\ te = 0 \end{smallmatrix} \right\rangle &
\end{tikzcd}$$

Satisfaction of Uppaal Queries by Configurations

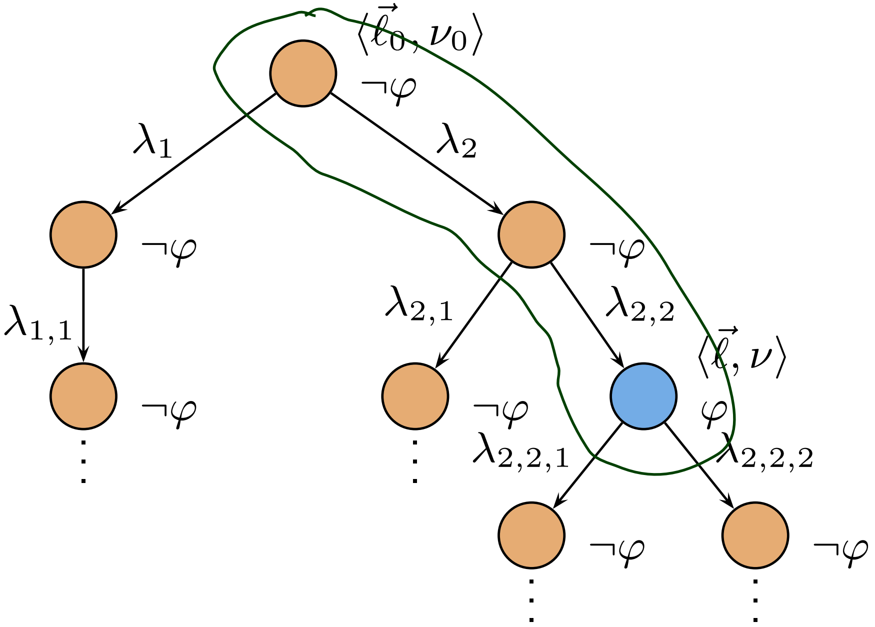
Exists finally:

ith configuration of ξ

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

"some configuration satisfying $term$ is reachable"

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

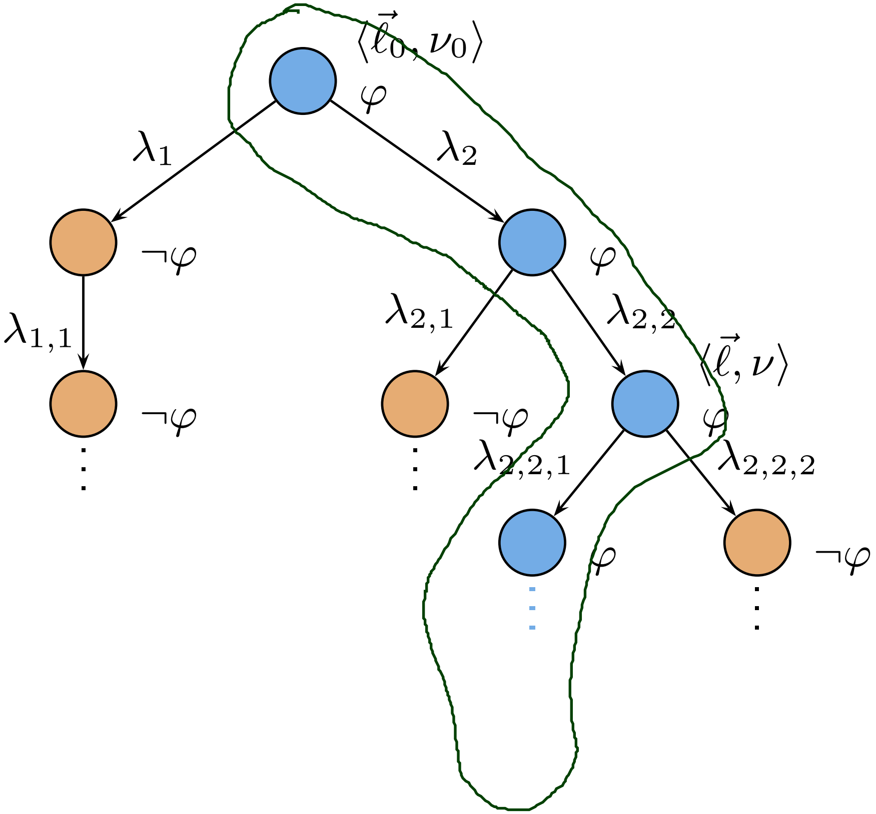


Satisfaction of Uppaal Queries by Configurations

Exists globally:

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

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



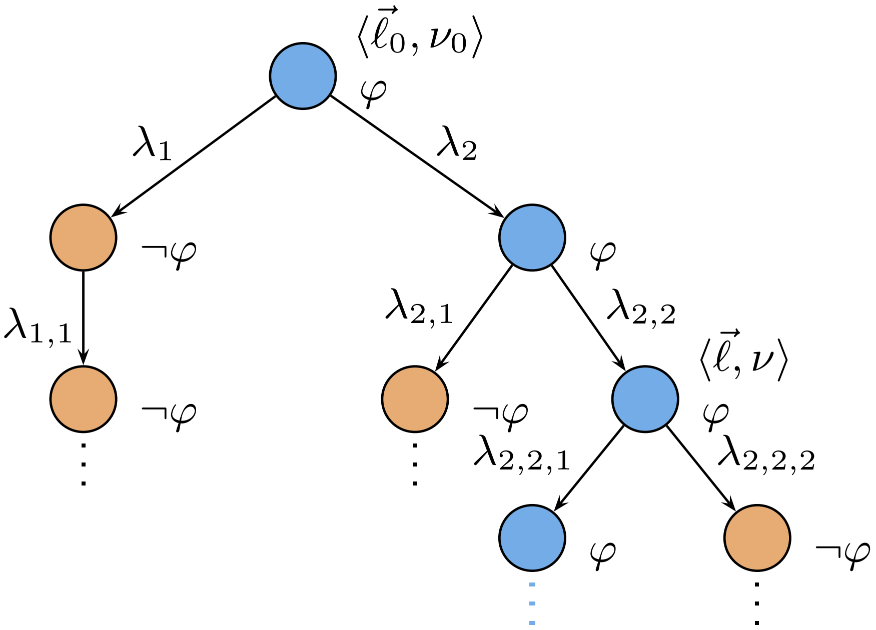
Satisfaction of Uppaal Queries by Configurations

Exists globally:

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

“on some computation path, all configurations satisfy $term$”

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



Satisfaction of Uppaal Queries by Configurations

- **Always globally**:
  - $\langle \vec{\ell}_0, \nu_0 \rangle \models \forall\Box\, term$ iff $\langle \vec{\ell}_0, \nu_0 \rangle \not\models \exists\Diamond\, \neg term$

    "not (some configuration satisfying $\neg term$ is reachable)"
    or: "all reachable configurations satisfy $term$"

- **Always finally**:
  - $\langle \vec{\ell}_0, \nu_0 \rangle \models \forall\Diamond\, term$ iff $\langle \vec{\ell}_0, \nu_0 \rangle \not\models \exists\Box\, \neg term$

    "not (on some computation path, all configurations satisfy $\neg term$)"
    or: "on all computation paths, there is a configuration satisfying $term$"

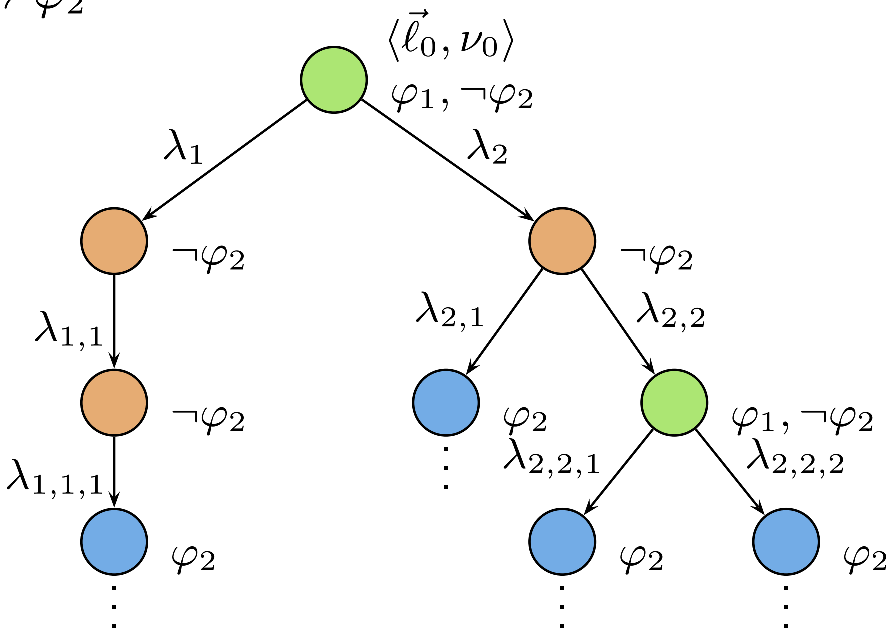
Satisfaction of Uppaal Queries by Configurations

Leads to:

- $\langle \vec{\ell}_0, \nu_0 \rangle \models term_1 \longrightarrow term_2$ iff $\forall$ path ξ of $\mathcal{N}$ starting in $\langle \vec{\ell}_0, \nu_0 \rangle$ $\forall\, i \in \mathbb{N}_0 \bullet$
$\xi^i \models term_1 \implies \xi^i \models \forall\Diamond\, term_2$

"on all paths, from each configuration satisfying $term_1$,
a configuration satifying $term_2$ is reachable" (**response pattern**)

Example: $\langle \vec{\ell}_0, \nu_0 \rangle \models \varphi_1 \longrightarrow \varphi_2$



CFA Model-Checking

Definition. Let $\mathcal{N} = \mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ be a network and F a query.

(i) We say $\mathcal{N}$ **satisfies** F, denoted by $\mathcal{N} \models F$, if and only if $C_{ini} \models F$.

(ii) The **model-checking problem** for $\mathcal{N}$ and F is to decide whether $(\mathcal{N}, F) \in \models$.

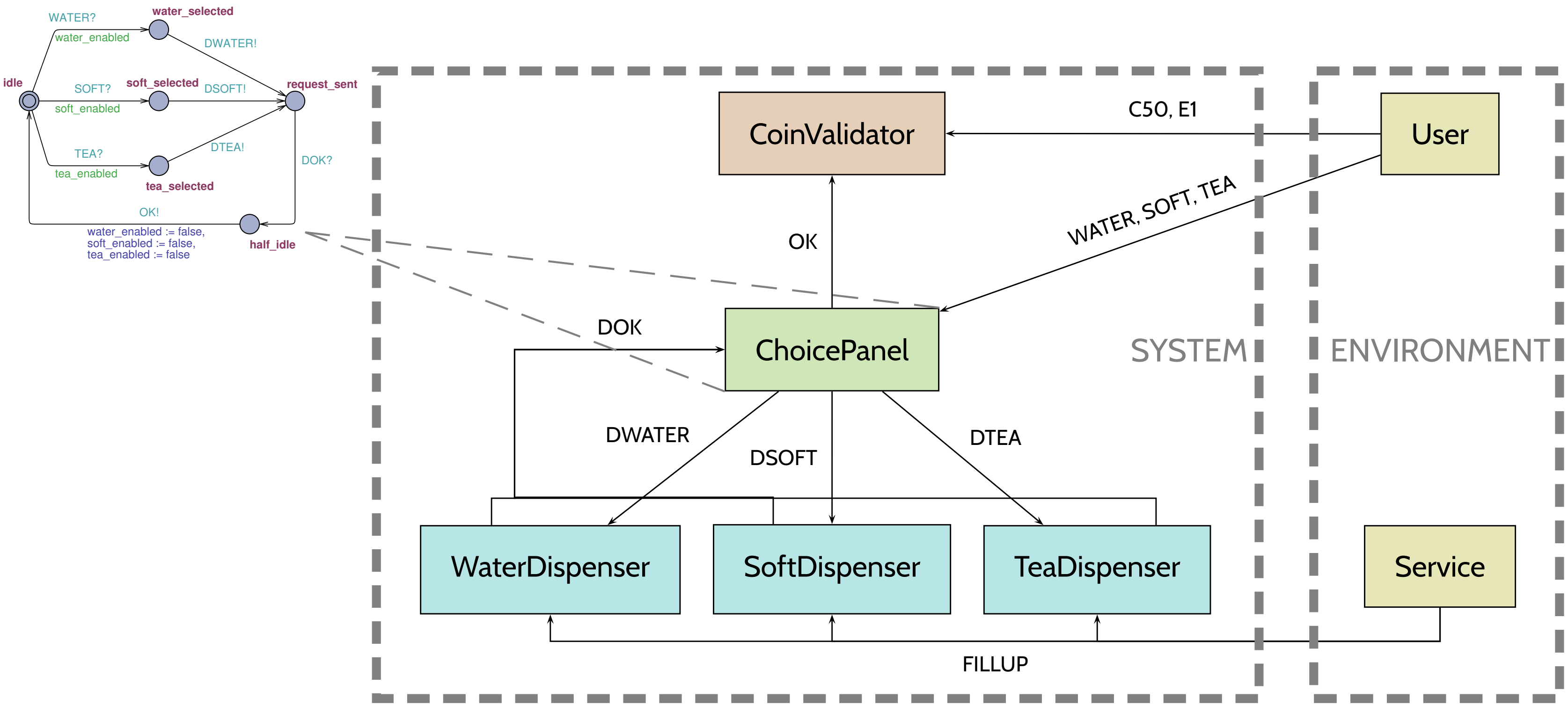
Proposition.
The model-checking problem for communicating finite automata is **decidable**.

Content

- **Design Patterns**
  - Strategy, Examples
- **Communicating Finite Automata** (CFA)
  - concrete and abstract syntax,
  - networks of CFA,
  - operational semantics.
- **Transition Sequences**
- **Deadlock, Reachability**
- **Uppaal**
  - tool demo (simulator),
  - query language,
  - CFA model-checking.
- **CFA at Work**
  - drive to configuration, scenarios, invariants
  - tool demo (verifier).
- **CFA vs. Software**

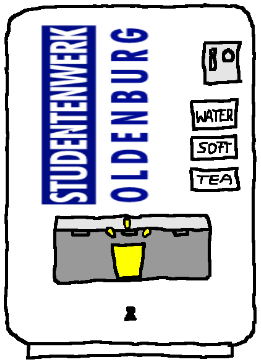
CFA and Queries at Work

Model Architecture — Who Talks What to Whom



- **Shared variables**:
  - `bool water_enabled, soft_enabled, tea_enabled;`
  - `int w = 3, s = 3, t = 3;`
- **Note**: Our model does not use scopes (“information hiding”) for channels.
  That is, ‘Service’ could send ‘WATER’ if the modeler wanted to.

Design Sanity Check: Drive to Configuration



- **Question**: Is 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

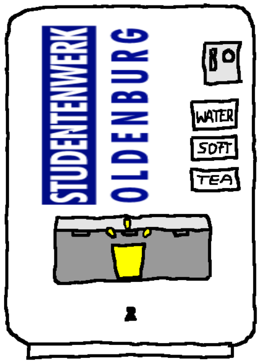
$$w = 0$$

is reachable, i.e. check

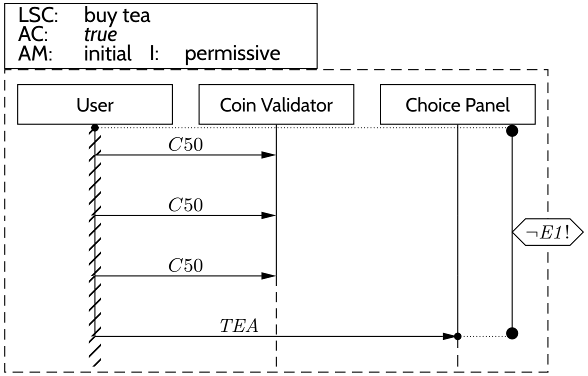
$$\mathcal{N}_{\mathrm{VM}} \models \exists\Diamond\, w = 0.$$

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

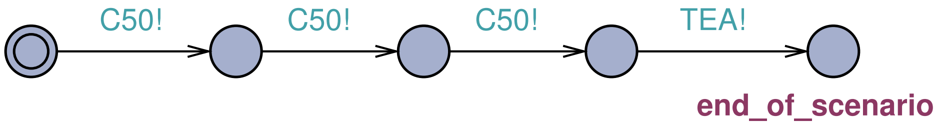
Design Check: Scenarios



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



- **Approach**: Use the following newly created CFA ‘Scenario’

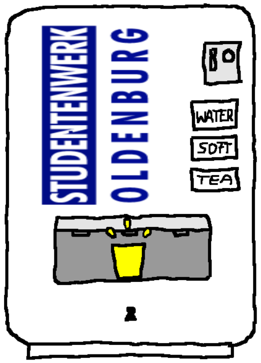


instead of **User** and check whether location `end_of_scenario` is reachable, i.e. check

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

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.)

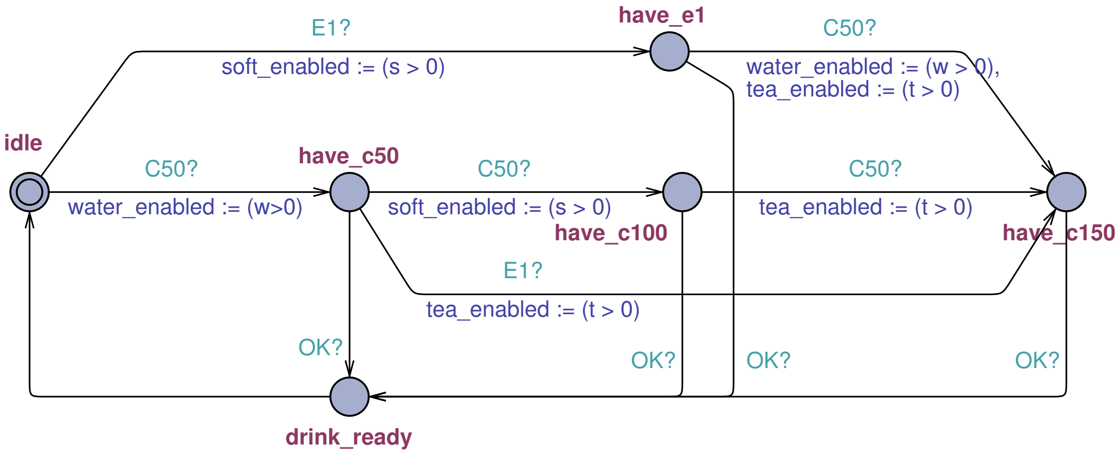
- **Approach**: Check whether the implication

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

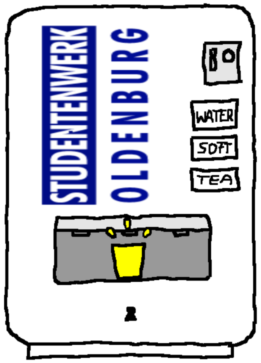
holds in all reachable configurations, i.e. check

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

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



Design Verification: Sanity Check



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

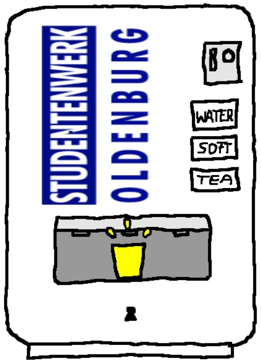
$$\texttt{tea\_enabled} \implies \text{CoinValidator}.\texttt{have\_c150}$$

  holds vacuously.)

- **Approach**: Check whether a configuration satisfying `water_enabled` $= 1$ is reachable.

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

Design Verification: Another Invariant



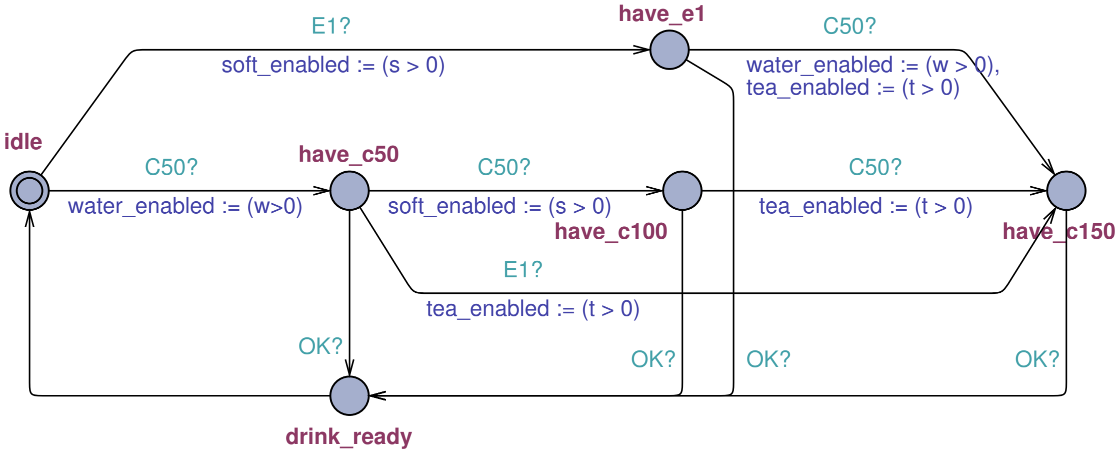
- **Question**: Is it the case that, if there is money in the machine and water in stock, that the "water" button is enabled?

- **Approach**: Check

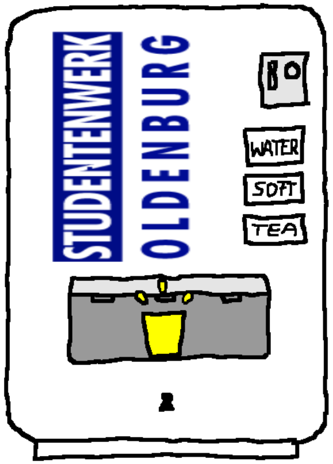
$$\mathcal{N}_{\mathrm{VM}} \models \forall\Box\,(\text{CoinValidator}.\texttt{have\_c50}\ \texttt{or}\ \text{CoinValidator}.\texttt{have\_c100}\ \texttt{or}\ \text{CoinValidator}.\texttt{have\_c150})$$

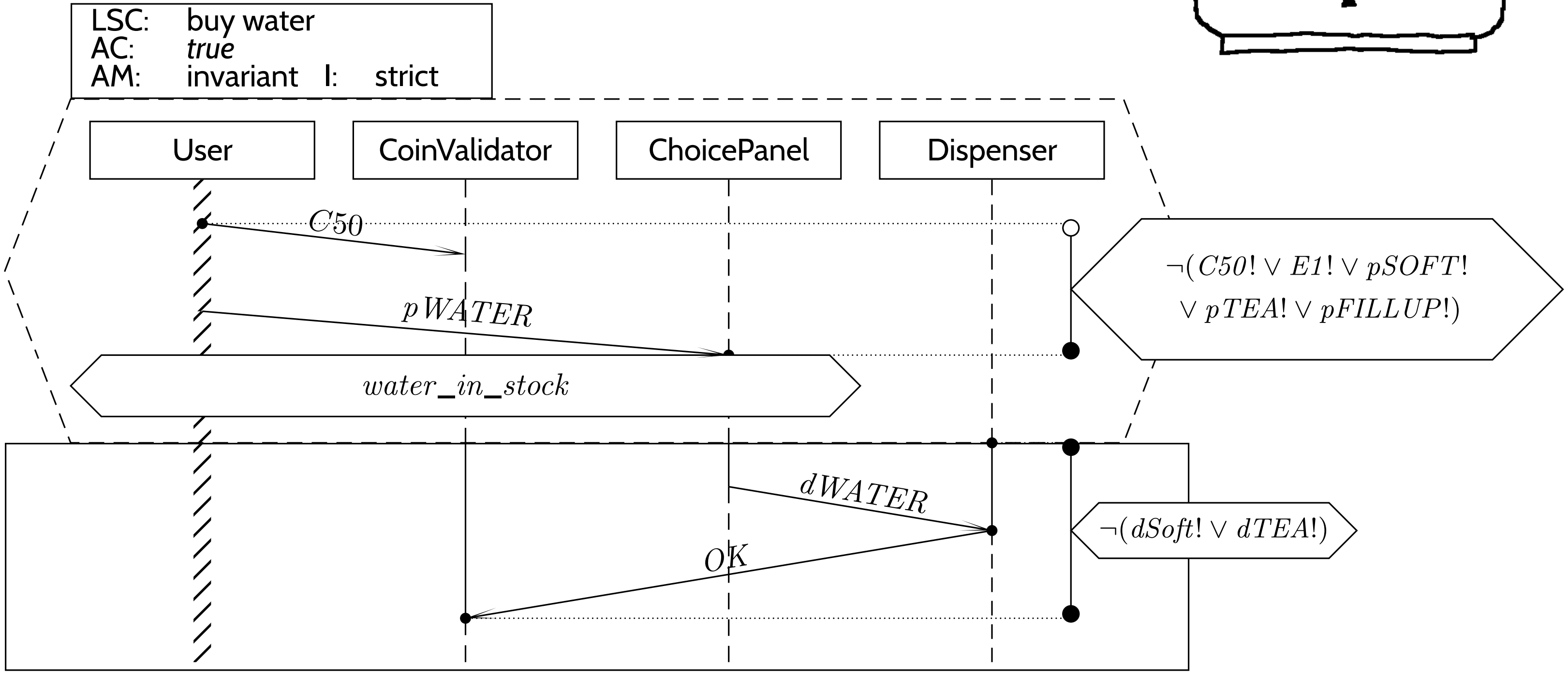
$$\texttt{imply water\_enabled}.$$

and $w > 0$



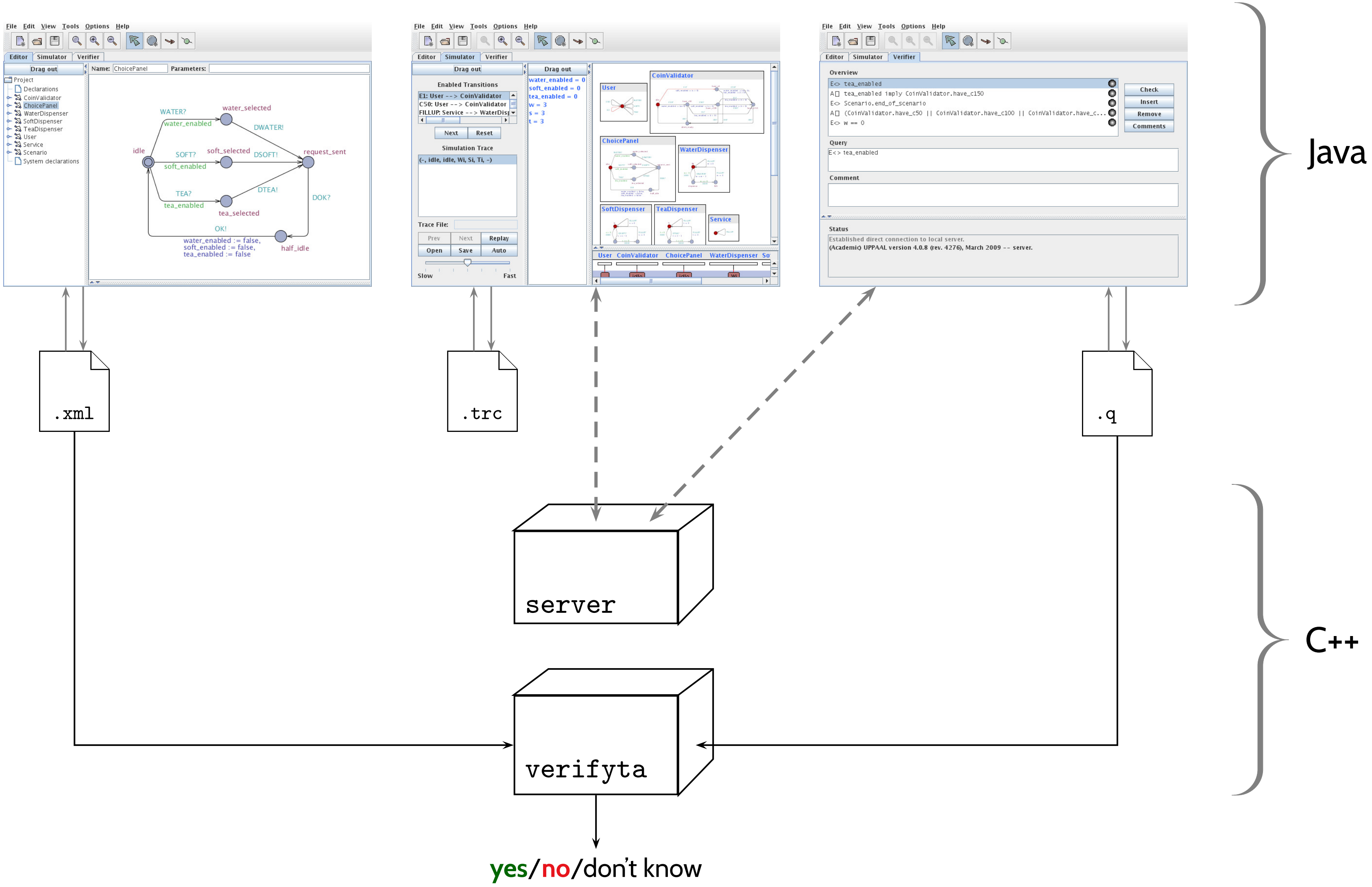
Recall: Universal LSC Example





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

Uppaal Architecture



Content

- **Design Patterns**
  - Strategy, Examples
- **Communicating Finite Automata** (CFA)
  - concrete and abstract syntax,
  - networks of CFA,
  - operational semantics.
- **Transition Sequences**
- **Deadlock, Reachability**
- **Uppaal**
  - tool demo (simulator),
  - query language,
  - CFA model-checking.
- **CFA at Work**
  - drive to configuration, scenarios, invariants
  - tool demo (verifier).
- **CFA vs. Software**

Tell Them What You've Told Them…

- A **network of communicating finite automata**
  - describes a **labelled transition system**,
  - can be used to **model** software behaviour.
- The **Uppaal Query Language** can be used to
  - formalize **reachability** ($\exists\Diamond\ CF$, $\forall\Box\ CF$, …) and
  - **leadsto** ($CF_1 \longrightarrow CF_2$) properties.
- Since the **model-checking problem** of CFA is **decidable**,
  - there are tools which **automatically check** whether a network of CFA satisfies a given query.
- Use model-checking, e.g., to
  - **obtain a computation path** to a certain configuration (**drive-to-configuration**),
  - check whether a **scenario** is possible,
  - check whether an **invariant** is satified.

    (If not, analyse the design further using the obtained **counter-example**).

References

References

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language - Towns, Buildings, Construction*. Oxford University Press.

Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on uppaal 2004-11-17. Technical report, Aalborg University, Denmark.

Gamma, E., Helm, R., Johnsson, R., and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

JHotDraw (2007). `http://www.jhotdraw.org`.

Larsen, K. G., Pettersson, P., and Yi, W. (1997). UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152.

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

Olderog, E.-R. and Dierks, H. (2008). *Real-Time Systems - Formal Specification and Automatic Verification*. Cambridge University Press.