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

- Introduction and Vocabulary
- Software Modelling I
  1. views and viewpoints, the 4+1 view
  2. model-driven/-based software engineering
  3. Modelling structure
     a) (simplified) class diagrams
     b) (simplified) object diagrams
     c) (simplified) object constraint logic (OCL)
     d) Unified Modelling Language (UML)

- Principles of Design
  1. modularity, separation of concerns
  2. information hiding and data encapsulation
  3. abstract data types, object orientation
  4. Design Patterns

- Software Modelling II
  1. Modelling behaviour
     a) communicating finite automata
     b) Uppaal query language
     c) basic state-machines
     d) an outlook on hierarchical state-machines
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
• In a sense the same as architectural patterns, but on a lower scale.
• Often traced back to (Alexander et al., 1977; Alexander, 1979).

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

### Example: Strategy

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

#### Structure

```plaintext
StrategyContext
  + contextInterface()

Strategy
  - algorithm()

ConcreteStrategy1
  + algorithm()

ConcreteStrategy2
  + algorithm()
```

### Example: Pattern Usage and Documentation

Pattern usage in JHotDraw framework ([HotDraw, 2007](#)) ([Diagram: (Ludewig and Lichter, 2013)](#))

#### Strategy

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

### 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. |
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.
“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.
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
Communicating Finite Automata

presentation follows (Olderog and Dierks, 2008)
**Channel Names and Actions**

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

- A set \((a, b \in \text{Chan})\) of **channel names** or **channels**.

- For each channel \(a \in \text{Chan}\), two **visible actions**: \(a?\) and \(a!\) denote **input** and **output** on the channel \((a?, a! \notin \text{Chan})\).

- \(\tau \notin \text{Chan}\) represents an **internal action**, not visible from outside.

- \((\alpha, \beta \in \text{Act}) := \{a? \mid a \in \text{Chan}\} \cup \{a! \mid a \in \text{Chan}\} \cup \{\tau\}\) is the set of **actions**.

- An alphabet \(B\) is a set of **channels**, i.e. \(B \subseteq \text{Chan}\).

- 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:** \(\text{Chan}_{??} = \text{Act}\).
Let \((v, w \in V)\) be a set of \((\text{finite domain})\) integer variables.

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

- A modification on \(v\) is of the form
  \[ v := \varphi, \quad v \in V, \quad \varphi \in \Psi(V). \]

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 \textbf{reset vector} (or \textbf{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

\[ A = (L, B, V, E, \ell_{\text{ini}}) \]

where

- \((\ell \in L)\) \(L\) is a finite set of \textbf{locations} (or \textbf{control states}).
- \(B \subseteq \text{Chan}\),
- \(V\): a set of data variables,
- \(E \subseteq L \times B \times \Phi(V) \times R(V)^* \times L\): a finite set of \textbf{directed edges} such that
  \((\ell, \alpha, \varphi, \vec{r}, \ell') \in E \land \text{chan}(\alpha) \in U \implies \varphi = \text{true}.\)
  Edges \((\ell, \alpha, \varphi, \vec{r}, \ell')\) from location \(\ell\) to \(\ell'\) are labelled with an \textbf{action} \(\alpha\), a guard \(\varphi\), and a list \(\vec{r}\) of \textbf{modifications}.
- \(\ell_{\text{ini}}\) is the \textbf{initial location}. 

Operational Semantics of Networks of CFA

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

The operational semantics of the network of CFA $\mathcal{C}(A_1, \ldots, A_n)$ is the labelled transition system

$$T(\mathcal{C}(A_1, \ldots, A_n)) = (Conf, Chan \cup \{\tau\}, \lambda, \rightarrow, \lambda \in Chan \cup \{\tau\}, C_{ini})$$

where
- $V = \bigcup_{i=1}^{n} V_i$ is the set of variables,
- $Conf = \{ (\ell, \nu) | \ell_i \in L_i, \nu : V \rightarrow 2^V \}$,
- $C_{ini} = (\vec{\ell}_{ini}, \nu_{ini})$ with $\nu_{ini}(v) = 0$ for all $v \in V$.

The transition relation consists of transitions of the following two types.
• \( \nu : V \to \mathcal{D}(V) \) is a **valuation** of the variables.

• A valuation \( \nu \) of the variables canonically assigns an integer value \( \nu(\varphi) \) to each integer expression \( \varphi \in \Phi(V) \).

• \( \models \subseteq (V \to \mathcal{D}(V)) \times \Phi(V) \) is the canonical **satisfaction relation** between valuations and integer expressions from \( \Phi(V) \).

• Effect of modification \( r \in R(V) \) on \( \nu \), denoted by \( \nu[r] \):
  \[
  \nu[r]_v := \begin{cases} 
  \nu(\varphi), & \text{if } a = v, \\
  \nu(a), & \text{otherwise} 
  \end{cases}
  \]

• 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** \( \langle \ell, \nu \rangle \xrightarrow{\tau} \langle \ell', \nu' \rangle \) occurs if there is \( i \in \{1, \ldots, n\} \) and
  • there is a \( \tau \)-edge \( (\ell_i, \tau, \varphi, \vec{r}, \ell'_i) \in E_i \) such that
    • \( \nu \models \varphi \),  
      “source valuation satisfies guard”
    • \( \vec{\ell} = \ell_i :\ell'_i \),  
      “automaton \( i \) changes location”
    • \( \nu' = \nu[\vec{r}] \),  
      “\( \nu' \) is the result of applying \( \vec{r} \) on \( \nu \)”

• A **synchronisation transition** \( \langle \vec{\ell}, \nu \rangle \xrightarrow{b} \langle \vec{\ell}', \nu' \rangle \) occurs if there are \( i, j \in \{1, \ldots, n\} \) with \( i \neq j \) and
  • there are edges \( (\ell_i, b! , \varphi_i, \vec{r}_i, \ell'_i) \in E_i \) and \( (\ell_j, b? , \varphi_j, \vec{r}_j, \ell'_j) \in E_j \) such that
    • \( \nu \models \varphi_i \land \varphi_j \),  
      “source valuation satisfies guards (\\!)”
    • \( \vec{\ell} = \ell_i :\ell'_i | \ell_j :\ell'_j \),  
      “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 \( \nu \)”

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

- A transition sequence of $C(A_1, \ldots, A_n)$ is any (in)finite sequence of the form

$$\langle \ell_0, \nu_0 \rangle \xrightarrow{\lambda_1} \langle \ell_1, \nu_1 \rangle \xrightarrow{\lambda_2} \langle \ell_2, \nu_2 \rangle \xrightarrow{\lambda_3} \ldots$$

with

- $\langle \ell_0, \nu_0 \rangle = C_{\text{ini}}$.
- for all $i \in \mathbb{N}$, there is $\lambda_{i+1}$ in $T(C(A_1, \ldots, A_n))$ with $\langle \ell_i, \nu_i \rangle \xrightarrow{\lambda_{i+1}} \langle \ell_{i+1}, \nu_{i+1} \rangle$. 

---

**Example**

**ChoicePanel:** (simplified)

![Diagram of the ChoicePanel](Image)

**User:**

![Diagram of the User](Image)
Deadlock

- A configuration \((\bar{\ell}, \nu)\) of \(C(A_1, \ldots, A_n)\) is called deadlock if and only if there are no transitions from \((\bar{\ell}, \nu)\), i.e. if
  \[
  \neg \exists \lambda \in \Lambda \exists (\bar{\ell}', \nu') \in \text{Conf} \cdot (\bar{\ell}, \nu) \xrightarrow{\lambda} (\bar{\ell}', \nu').
  \]
  The network \(C(A_1, \ldots, A_n)\) is said to have a deadlock if and only if there is a reachable configuration \((\bar{\ell}, \nu)\) which is a deadlock.

Reachability

- A configuration \((\bar{\ell}, \nu)\) is called reachable (in \(C(A_1, \ldots, A_n)\)) from \((\bar{\ell}_0, \nu_0)\) if and only if there is a transition sequence of the form
  \[
  (\bar{\ell}_0, \nu_0) \xrightarrow{\lambda_1} (\bar{\ell}_1, \nu_1) \xrightarrow{\lambda_2} (\bar{\ell}_2, \nu_2) \xrightarrow{\lambda_3} \ldots \xrightarrow{\lambda_n} (\bar{\ell}_n, \nu_n) = (\bar{\ell}, \nu).
  \]

- A configuration \((\bar{\ell}, \nu)\) is called reachable (without "from") if and only if it is reachable from \(C_{\text{init}}\).

- A location \(\ell \in L_i\) is called reachable if and only if any configuration \((\bar{\ell}, \nu)\) with \(\ell_i = \ell\) is reachable, i.e. there exist \(\ell'\) and \(\nu\) such that \(\ell_i = \ell\) and \((\bar{\ell}, \nu)\) is reachable.
Uppaal

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

Tool Demo
Consider $\mathcal{N} = C(A_1, \ldots, A_n)$ over data variables $V$.

- **basic formula:**
  
  $\text{atom} ::= A_i.\ell \mid \varphi \mid \text{deadlock}$
  
  where $\ell \in L_i$ is a location and $\varphi$ an expression over $V$.

- **configuration formulae:**
  
  $\text{term} ::= \text{atom} \mid \text{not term} \mid \text{term}_1 \text{ and term}_2$

- **existential path formulae:**
  
  $\text{e-formula} ::= \exists \Diamond \text{term} \quad \text{(exists finally)}$
  
  $\mid \exists \Box \text{term} \quad \text{(exists globally)}$

- **universal path formulae:**
  
  $\text{a-formula} ::= \forall \Diamond \text{term} \quad \text{(always finally)}$
  
  $\mid \forall \Box \text{term} \quad \text{(always globally)}$
  
  $\mid \text{term}_1 \rightarrow \text{term}_2 \quad \text{(leads to)}$

- **formulae (or queries):**
  
  $F ::= \text{e-formula} \mid \text{a-formula}$

---

**Satisfaction of Uppaal Queries by Configurations**

- The satisfaction relation
  
  $(\vec{\ell}, \nu) \models F$

  between configurations
  
  $(\vec{\ell}, \nu) = ((\ell_1, \ldots, \ell_n), \nu)$

  of a network $C(A_1, \ldots, A_n)$ and formulae $F$ of the Uppaal logic

  is defined **inductively** as follows:

  - $(\vec{\ell}, \nu) \models \text{deadlock}$
    
    iff $(\vec{\ell}, \nu)$ is a deadlock configuration.

  - $(\vec{\ell}, \nu) \models A_i.\ell$
    
    iff $\ell_i = \ell$

  - $(\vec{\ell}, \nu) \models \varphi$
    
    iff $\nu \models \varphi$

  - $(\vec{\ell}, \nu) \models \text{not term}$
    
    iff $(\vec{\ell}, \nu) \not\models \text{term}$

  - $(\vec{\ell}, \nu) \models \text{term}_1 \text{ and term}_2$
    
    iff $(\vec{\ell}, \nu) \models \text{term}_1$ and $(\vec{\ell}, \nu) \models \text{term}_2$
Example: Computation Paths vs. Computation Tree

ChoicePanel:

User:

Example: Computation Paths vs. Computation Graph

(or: Transition Graph)
Satisfaction of Uppaal Queries by Configurations

**Exists finally:**
- \( (\vec{e}_0, \nu_0) \models \exists ♦ \text{term} \) iff \( \exists \text{path } ξ \text{ of } N \text{ starting in } (\vec{e}_0, \nu_0) \exists i \in N_0 \cdot ξ_i \models \text{term} \)

"some configuration satisfying term is reachable"

**Example:** \( (\vec{e}_0, \nu_0) \models \exists ♦ \varphi \)


Satisfaction of Uppaal Queries by Configurations

**Exists globally:**
- \( (\vec{e}_0, \nu_0) \models \exists □ \text{term} \) iff \( \exists \text{path } ξ \text{ of } N \text{ starting in } (\vec{e}_0, \nu_0) \forall i \in N_0 \cdot ξ_i \models \text{term} \)

**Example:** \( (\vec{e}_0, \nu_0) \models \exists □ \varphi \)
Satisfaction of Uppaal Queries by Configurations

**Exists globally:**
- \( \langle \vec{v}_0, \nu_0 \rangle |\exists \square term \) iff \( \exists \) path \( \xi \) of \( \mathcal{N} \) starting in \( \langle \vec{v}_0, \nu_0 \rangle \)
  \( \forall i \in \mathbb{N}_0 \! \cdot \! \xi^i |\satisfies term \)

"on some computation path, all configurations satisfy \( term \)"

**Example:** \( \langle \vec{v}_0, \nu_0 \rangle |\exists \Box \varphi \)

*Satisfaction of Uppaal Queries by Configurations*

- **Always globally:**
  - \( \langle \vec{v}_0, \nu_0 \rangle |\forall \square term \) iff \( \langle \vec{v}_0, \nu_0 \rangle \not|\exists \Diamond \neg term \)

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

- **Always finally:**
  - \( \langle \vec{v}_0, \nu_0 \rangle |\forall \Diamond term \) iff \( \langle \vec{v}_0, \nu_0 \rangle \not|\exists \Box \neg term \)

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

- \( \langle \vec{e}_0, \nu_0 \rangle \models term_1 \rightarrow term_2 \) iff \( \forall \text{ path } \xi \text{ of } \mathcal{N} \text{ starting in } \langle \vec{e}_0, \nu_0 \rangle \forall i \in \mathbb{N}_0 \cdot \xi^i \models term_1 \implies \xi^i \models \forall \emptyset term_2 \)

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

Example: \( \langle \vec{e}_0, \nu_0 \rangle \models \varphi_1 \rightarrow \varphi_2 \)

---

**CFA Model-Checking**

**Definition.** Let \( \mathcal{N} = \mathcal{C}(A_1, \ldots, A_n) \) 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 it (at all) possible to have no water in the vending machine model? (Otherwise, the design is definitely broken.)

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

\[ w = 0 \]

is reachable, i.e. check

\[ N_{VM} \models \exists \diamond w = 0. \]

for the vending machine model \( N_{VM} \).
**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

\[ N_{VM}' |= \exists \text{Scenario.end_of_scenario}. \]

for the modified vending machine model \( N_{VM}' \).

---

**Design Verification: Invariants**

- **Question**: Is it the case that the "tea" button is only enabled if there is €1.50 in the machine? (Otherwise, the design is broken.)

- **Approach**: Check whether the implication

\[ \text{tea_enabled} \implies \text{CoinValidator.have_c150} \]

holds in all reachable configurations, i.e. check

\[ N_{VM} |= \forall \Box \text{tea_enabled imply CoinValidator.have_c150} \]

for the vending machine model \( N_{VM} \).
**Design Verification: Sanity Check**

- **Question**: Is the “tea” button ever enabled? (Otherwise, the considered invariant \( \text{tea\_enabled} \implies \text{CoinValidator\_have\_c150} \) holds vacuously.)

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

\[
\mathcal{N}_{VM} \models \forall (\text{CoinValidator\_have\_c50} \lor \text{CoinValidator\_have\_c100} \lor \text{CoinValidator\_have\_c150}) \implies \text{water\_enabled}.
\]
Recall: Universal LSC Example

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

#### User
- **CoinValidator**
- **ChoicePanel**
- **Dispenser**

$$\neg (C_{50} \lor E_{1} \lor p_{SOFT} \lor p_{TEA} \lor p_{FILLUP})$$

$$d_{WATER} = OK \land \neg (d_{Soft} \lor d_{TEA})$$

### Uppaal Architecture

#### Java
- **server**
- **verifyta**

#### C++
- yes/no
- don't know
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 ($\exists ♦ CF, ∀ □ CF, …$) and
  - leadsto ($CF_1 \rightarrow CF_2$) 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 satisfied.
    (If not, analyse the design further using the obtained counter-example).
References