Contents & Goals

Last Lecture:
- requirements engineering basics, “the natural language case”

This Lecture:
- **Educational Objectives**: Capabilities for following tasks/questions.
  - What is a rule, a decision table?
  - What is the interleaving, collecting, update semantics?
  - Analyse this rule for: vacuous, redundant, complete, consistent
  - In what sense can decision tables serve as requirements specification language?
  - Formalise this requirement with a decision table.
  - What does this have to do with the previous lecture?

- **Content**:
  - definition decision table syntax and interleaving semantics,
  - definition tables as requirements specification,
  - interesting/useful properties of decision tables
Recall: Formal Methods

**Definition.** [Bjørner and Havelund, 2014] A method is called a formal method if and only if its techniques and tools can be explained in mathematics.

**Example:** If a method includes, as a tool, a specification language, then that language has
- a formal syntax,
- a formal semantics, and
- a formal proof system.

---

**Formal, Rigorous, or Systematic Development**

- The techniques of a formal method help
  - construct a specification, and/or
  - analyse a specification, and/or
  - transform (refine) one (or more) specification(s) into a program.

The techniques of a formal method, (besides the specification languages) are typically software packages that help developers use the techniques and other tools.

The aim of developing software, either
- formally (all arguments are formal) or
- rigorously (some arguments are made and they are formal) or
- systematically (some arguments are made in a form that can be made formal)

is to (be able to) reason in a precise manner about properties of what is being developed. ([Bjørner and Havelund, 2014])

---

Decision tables (DT) are one example for a formal requirements specification language:
- we give a formal syntax and semantics,
- requirements quality criteria, e.g. completeness, can be formally defined,
- thus for a DT we can formally argue whether it is complete or not,
- (some) formal arguments can be done automatically (→ tool support).

---

... so, off to “technological paradise” where [...] everything happens according to the blueprints”.

([Kopetz, 2011; Lovins and Lovins, 2001])
**Decision Tables**

**Definition.** [Decision Table] Let $C$ be a set of (atomic) **conditions** and $A$ a set of **actions**.

(i) The set $\Phi(C)$ of **premises** over $C$ consists of the terms defined by the following grammar: $\varphi ::= \text{true} \mid c \mid \neg \varphi_1 \mid \varphi_1 \lor \varphi_2$, $c \in C$.

(ii) A **rule** $r$ (over $C$ and $A$) is a pair $(\varphi, \alpha)$, written $\varphi \rightarrow \alpha$, which comprises

- a **premise** $\varphi \in \Phi(C)$ and
- a **finite set** $\alpha \subseteq A$ of actions (the **effect**).

(iii) Any finite set $T$ of rules (over $C$ and $A$) is called **decision table** (over $C$ and $A$).
**Decision Tables: Example**

This might’ve been the specification of lecture hall 101-0-026’s ventilation system:

- **Conditions:**
  
  \[ C = \{ \text{button\_pressed, ventilation\_on, ventilation\_off} \} \]  
  
  **shorthands:** \{b, on, off\}.

- **Actions:**

  \[ A = \{ \text{do\_ventilate, stop\_ventilate} \} \]  
  
  **shorthands:** \{go, stop\}.

- **Rules:**

  \[ r_1 = b \land \neg \text{off} \rightarrow \{ \text{go} \} \]

  \[ r_2 = b \land \text{on} \rightarrow \{ \text{stop} \} \]

- **Decision table:**

  \[ T = \{ r_1, r_2 \}. \]

---

**And Where’s The Table?**

Decision tables can be written in tabular form:

<table>
<thead>
<tr>
<th>( T ): room ventilation</th>
<th>( r_1 )</th>
<th>( r_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td>button pressed?</td>
<td>( \times )</td>
</tr>
<tr>
<td>( \text{off} )</td>
<td>ventilation off?</td>
<td>( \times )</td>
</tr>
<tr>
<td>( \text{on} )</td>
<td>ventilation on?</td>
<td>( _ )</td>
</tr>
<tr>
<td>( \text{go} )</td>
<td>start ventilation</td>
<td>( \times )</td>
</tr>
<tr>
<td>( \text{stop} )</td>
<td>stop ventilation</td>
<td>( _ )</td>
</tr>
</tbody>
</table>

From the table to the rules:

- \( r_1 = b \land \text{on} \land \neg \text{off} \rightarrow \{ \text{go} \} \)
- \( r_2 = b \land \neg \text{on} \land \text{off} \rightarrow \{ \text{stop} \} \)
And Where’s The Table?

Decision tables can be written in tabular form:

<table>
<thead>
<tr>
<th>$T$: room ventilation</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$ button pressed?</td>
<td>×</td>
<td>×</td>
<td>⊗</td>
</tr>
<tr>
<td>$off$ ventilation off?</td>
<td>×</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$on$ ventilation on?</td>
<td>-</td>
<td>×</td>
<td>-</td>
</tr>
<tr>
<td>$go$ start ventilation</td>
<td>×</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$stop$ stop ventilation</td>
<td>-</td>
<td>×</td>
<td>-</td>
</tr>
</tbody>
</table>

From the table to the rules:
1. $r_1 = b \land on \land \neg off \rightarrow \{go\}$
2. $r_2 = b \land \neg on \land off \rightarrow \{stop\}$
3. $r_3 = \neg on \land \neg off \rightarrow \emptyset$

Decision Tables vs. Rules In General

<table>
<thead>
<tr>
<th>$T$: decision table</th>
<th>$r_1$</th>
<th>...</th>
<th>$r_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$ description condition 1</td>
<td>$v_{1,1}$</td>
<td>...</td>
<td>$v_{1,n}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$c_m$ description condition $m$</td>
<td>$v_{m,1}$</td>
<td>...</td>
<td>$v_{m,n}$</td>
</tr>
<tr>
<td>$a_1$ description action 1</td>
<td>$w_{1,1}$</td>
<td>...</td>
<td>$w_{1,n}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$a_k$ description action $k$</td>
<td>$w_{k,1}$</td>
<td>...</td>
<td>$w_{k,n}$</td>
</tr>
</tbody>
</table>

$v_{i,j} \in \{-, \times, *, \}$, $w_{i,j} \in \{-, \times\}$

- $C = \{c_1, \ldots, c_m\}$, $A = \{a_1, \ldots, a_k\}$
- $r_i = F(v_{1,i}, c_i) \land \cdots \land F(v_{m,i}, c_m) \rightarrow \{a_j \mid w_{j,i} = \times\}$
- $F(v, c) = \begin{cases} c & \text{, if } v = \times \\ \neg c & \text{, if } v = - \\ \text{true} & \text{, if } v = * \end{cases}$
- Recall: $\bigwedge_{1 \leq j \leq m} F(v_{j,i}, c_i) = \text{true by definition.}$
- $T = \{r_1, \ldots, r_n\}$ (multiple tables $T_1, \ldots, T_n$ denote one set of rules)
- From rules to table: use disjunctive normal form of $\varphi$. 

By the Way: Decision Tables as Business Rules

<table>
<thead>
<tr>
<th>T1</th>
<th>cash a cheque</th>
<th>r1</th>
<th>r2</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>credit limit exceeded?</td>
<td>×</td>
<td>×</td>
<td>-</td>
</tr>
<tr>
<td>c2</td>
<td>payment history ok?</td>
<td>×</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c3</td>
<td>overdraft &lt; 500 €?</td>
<td>-</td>
<td>×</td>
<td>-</td>
</tr>
<tr>
<td>a1</td>
<td>cash cheque</td>
<td>×</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>a2</td>
<td>do not cash cheque</td>
<td>-</td>
<td>×</td>
<td>-</td>
</tr>
<tr>
<td>a3</td>
<td>offer new conditions</td>
<td>×</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(Balzert, 2009)

- One customer session at the bank:

  $\sigma \xrightarrow[{a_1,a_3}]{} \sigma'$

  if $\sigma = \{c_1 \mapsto 1, c_2 \mapsto 1, c_3 \mapsto 0\}$.

- clerk checks database state $\sigma$,
- database says: credit limit exceeded over 500 €, but payment history ok,
- clerk cashes cheque but offers new conditions.

Decision Table Standard Semantics

- Let $C$ be a set of conditions and $A$ a set of actions.
- Let $\Sigma := (C \rightarrow B)$ be the set of valuations of $C$, $B := \{0, 1\}$.
- Let $\varphi \in \Phi(C)$ be a premise and $\sigma \in \Sigma$. Set:
  - $\sigma \models true$, for all $\sigma \in \Sigma$,
  - $\sigma \models c$, if and only if $\sigma(c) = 1$,
  - $\sigma \models \neg \varphi_1$, if and only if $\sigma \not\models \varphi$,
  - $\sigma \models \varphi_1 \lor \varphi_2$, if and only if $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$.

  Note: In the following, we may use $\land$, $\implies$, $\iff$ as abbreviations as usual.

- We call a rule $r = \varphi \rightarrow \alpha$ over $C$ and $A$ enabled in $\sigma$ if and only if $\sigma \models \varphi$.

- Let $T$ be a decision table over $C$ and $A$. The set $[T]_{\text{interleave}}$, the (standard) interleaving semantics/interpretation of $T$, consists of the finite or infinite computation paths

  $\pi = \sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \sigma_2 \cdots$, $\sigma_i \in \Sigma, i \in \mathbb{N}_0$,

  where $\forall i \in \mathbb{N}_0 \exists r = \varphi \rightarrow \alpha \in T \bullet \sigma_i \models \varphi \land a_{i+1} = \alpha$. 
Decision Tables: Example

Back to lecture hall 101-0-026’s ventilation system:

- \( C = \{ \text{button\_pressed}, \text{ventilation\_on}, \text{ventilation\_off} \} \)  
  shorthands: \( \{ b, \text{on}, \text{off} \} \).
- \( A = \{ \text{do\_ventilate, stop\_ventilate} \} \)  
  shorthands: \( \{ \text{go, stop} \} \).
- \( r_1 = b \land \text{ventilation\_off} \rightarrow \{ \text{go} \} \), \( r_2 = b \land \text{ventilation\_on} \rightarrow \{ \text{stop} \} \), \( T = \{ r_1, r_2 \} \).

What’s in \([T]_{\text{interleaving}}\)? Naja, for example

- \( \pi_1 = \sigma_0 \xrightarrow{\{ \text{go} \}} \sigma_1 \xrightarrow{\{ \text{stop} \}} \sigma_2 \)
  \( \sigma_0 = \{ b \mapsto 1, \text{off} \mapsto 1, \text{on} \mapsto 0 \} \),
  \( \sigma_1 = \{ b \mapsto 1, \text{off} \mapsto 0, \text{on} \mapsto 1 \} \).
- \( \pi_2 = \sigma_0 \)
  \( \sigma_0 = \{ b \mapsto 0, \text{off} \mapsto 1, \text{on} \mapsto 0 \} \).
- \( \pi_3 = \sigma_0 \xrightarrow{\{ \text{go} \}} \sigma_1 \xrightarrow{\{ \text{go} \}} \sigma_2 \)
  \( \sigma_0 = \{ b \mapsto 1, \text{off} \mapsto 1, \text{on} \mapsto 0 \} \),
  \( \sigma_1 = \{ b \mapsto 1, \text{off} \mapsto 1, \text{on} \mapsto 0 \} \).
- \( \text{also } \pi_4 = \sigma_0 \xrightarrow{\{ \text{go} \}} \sigma_1 \xrightarrow{\{ \text{go} \}} \sigma_2 \ldots \)
  \( \sigma_i = \{ b \mapsto 1, \text{off} \mapsto 1, \text{on} \mapsto 0 \}, i \in \mathbb{N}_0 \).
- \( \text{but not } \sigma_0 \xrightarrow{\{ \text{go, stop} \}} \sigma_2 \)
  \( \sigma_0 = \{ b \mapsto 0, \text{off} \mapsto 1, \text{on} \mapsto 0 \} \).

Isn’t There a Bell Ringing...?

Definition. Software is a finite description \( S \) of a (possibly infinite) set \([S]\) of (finite or infinite) computation paths of the form

\[
\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \ldots
\]

where

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

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

\( \rightarrow \) a decision table \( T \) is software in the sense of the definition.

(surprise, surprise!?)
But We Want A Software Specification, Don’t We...?

- Let $T$ be a decision table over $C$ and $A$.
- Let $S$ be a software with $[S] = \{\sigma_0^s \xrightarrow{\alpha_1^1} \sigma_1^s \xrightarrow{\alpha_2^2} \sigma_2^s \cdots \} \in \Sigma^s$.
- Let $I : \Sigma^s \to (C \to B)$ be an interpretation of conditions $C$ in states $\Sigma^s$, and $M : A^* \to 2^A$ a mapping of events to sets of actions.
- We say $S$ implements $T$ wrt. $I$ and $M$ if and only if
  \[ \forall \sigma_0^s \xrightarrow{\alpha_1^1} \sigma_1^s \xrightarrow{\alpha_2^2} \sigma_2^s \cdots \in [S] \bullet I(\sigma_0^s) \xrightarrow{M(\alpha_1)} I(\sigma_1^s) \cdots \in \llbracket T \rrbracket_{\text{interleaving}} \]

- $T$ can be seen as a software specification by setting
  \[ [T]_{\text{spec}} := \{S \mid \exists I, M \bullet S \text{ implements } T \text{ wrt. } I \text{ and } M \}. \]

- Any software $S$, whose behaviour viewed through $I$ and $M$ is a subset (!) of $T$’s behaviour, satisfies the specification $T$.
- The computation paths of $T$ are all allowed for the final product $S$; what is not a computation path of $T$ is forbidden for the final product.

- The refinement view:
  A software $S'$ which is a refinement of a software $S \in [T]_{\text{spec}}$ satisfies specification $T$. 

...
**Decision Table as Requirements Specification: Examples**

How can these $I$ and $M$ look like?

**Example:**

- $\Sigma^1 = C \rightarrow \{0, 1\}$ — a state $\sigma \in \Sigma_1$ is a (boolean) valuation of the conditions;
  
  $I : \sigma^1 \rightarrow \sigma^1$, the identity

- $\Sigma^2 = \{b, V\} \rightarrow B \cup R_0^+$; $\sigma^2(b) \in B$ (button state), $\sigma^2(V) \in R_0^+$ (voltage at ventilator)
  
  $I$ is defined by:
  
  - $I(\sigma^2)(b) = 1$ if and only if $\sigma(b) = 1$,
  - $I(\sigma^2)(on) = 1$ if and only if $\sigma(V) \geq 0.7$ (ventilator rotates),
  - $I(\sigma^2)(off) = 1$ if and only if $\sigma(V) < 0.7$ (voltage too low for rotation).

- $\Sigma^3$: internal state of a Java VM running a Java program with global variables $b, on, off$.
  
  $I(\sigma^3)(b) = 1$ if and only if value of $b$ in $\sigma$ is Java’s `true`.

- In other words: $\Sigma^s$ can be everything, as long as you explain/define how to read out from $\sigma^s \in \Sigma^s$ whether the conditions (here: $b$, $on$, $off$) should be considered 0 or 1 in $\sigma$.

---

**Basic Requirements Quality Criteria for DTs**
Requirements on Requirements Specifications

A requirements specification should be

- **correct** — it correctly represents the wishes/needs of the customer,
- **complete** — all requirements (existing in somebody’s head, or a document, or . . .) should be present,
- **relevant** — things which are not relevant to the project should not be constrained,
- **consistent, free of contradictions** — each requirement is compatible with all other requirements; otherwise the requirements are not realisable,
- **neutral, abstract** — a requirements specification does not constrain the realisation more than necessary,
- **traceable, comprehensible** — the sources of requirements are documented, requirements are uniquely identifiable,
- **testable, objective** — the final product can objectively be checked for satisfying a requirement.

Requirements on Specifications and Formal Methods

- **correctness** is relative to “in the head of the customer” → still difficult;
- **complete**: we can at least define a kind of relative completeness in the sense of “did we cover all cases?”;
- **relevant** also not analyzable within decision tables;
- **consistency** can formally be analysed!
- **neutral/abstract** is relative to the realisation → still difficult;
  But formal requirements specification language tend to support abstract specifications; specifying technical details is tedious.
- **traceable/comprehensible** are meta-properties, need to be established separately;
- a formal requirements specification, e.g. using decision tables, is immediately objective/testable.

We can formally define additional quality criteria:

- rules should not be **useless** or **vacuous**,
- a **deterministic** decision table may be desired,
- rules should be **consistent** wrt. a domain model,
**Quality Criteria for DTs: Uselessness and Vacuity**

**Definition.** [Uselessness and Vacuity] Let $T$ be a decision table.

- A rule $r = \varphi \rightarrow \alpha$ is called **vacuous** (wrt. states $\Sigma$) if and only if there is no state $\sigma \in \Sigma$ such that $\sigma \models \varphi$.

- A rule $r = \varphi \rightarrow \alpha$ is called **useless** (or: **redundant**) if and only if there is another (different) rule $r'$ whose premise $\varphi'$ is subsumed by $\varphi$ and whose effect is the same as $r$'s, i.e. if

$$\exists r' = \varphi' \rightarrow \alpha', r' \neq r = \psi \models \varphi' \land \alpha = \alpha'.$$

$r'$ is called **subsumed** by $r$.

**Uselessness: Example**

**Example:**

- $(c \land \neg c) \rightarrow \alpha$ is **vacuous**.

**Proposition:** any rule with insatisfiable premise is vacuous.

<table>
<thead>
<tr>
<th>$T$: room ventilation</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$ button pressed?</td>
<td>×</td>
<td>×</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>off ventilation off?</td>
<td>×</td>
<td>-</td>
<td>*</td>
<td>-</td>
</tr>
<tr>
<td>on ventilation on?</td>
<td>-</td>
<td>×</td>
<td>*</td>
<td>-</td>
</tr>
<tr>
<td>go start ventilation</td>
<td>×</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>stop stop ventilation</td>
<td>-</td>
<td>×</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- $r_4$ is **useless** — it is subsumed by $r_3$.
- $r_3$ is **not subsumed** by $r_4$!
- **Proposition:** if rule $r$ is given in form of a table then $r$ is not vacuous (yet it may be subsumed by another rule).
Uselessness: Consequences

- **Doesn’t hurt** wrt. the final product:
  
  The decision table $T$ with useless rules has the same computation paths as the one with useless rules removed, thus specifies the same set of software.

- **But**
  
  - decision tables with useless rules are unnecessarily hard to work with (read, maintain, ...).
  
  - May make communication (with customer) **harder!**

---

Quality Criteria for DTs: Completeness

**Definition.** [Completeness] A decision table $T$ is called **complete** if and only if the disjunction of all rules’ premises is a **tautology**, i.e. if

$$
\models \bigvee_{\varphi \to \alpha \in T} \varphi.
$$
Completeness: Example

\[ \neg b \land \neg \text{on} \land \neg \text{off} \]

**is not complete:** there is no rule, e.g., for the case \( \neg b \land \text{on} \land \neg \text{off} \).

\[ b = 0, \, \text{off} = 0, \, \text{on} = 0 \]

• is complete.
Completeness: Example

<table>
<thead>
<tr>
<th>$T$: room ventilation</th>
<th>$r_1$</th>
<th>$r_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$ button pressed?</td>
<td>$\times$</td>
<td>$\times$</td>
</tr>
<tr>
<td>$\neg on$ ventilation off?</td>
<td>$\times$</td>
<td>$-$</td>
</tr>
<tr>
<td>$on$ ventilation on?</td>
<td>$-$</td>
<td>$\times$</td>
</tr>
<tr>
<td>$go$ start ventilation</td>
<td>$\times$</td>
<td>$-$</td>
</tr>
<tr>
<td>$stop$ stop ventilation</td>
<td>$-$</td>
<td>$\times$</td>
</tr>
</tbody>
</table>

- is not complete: there is no rule, e.g., for the case $\neg b \land on \land \neg off$.

Incompleteness: Consequences

- An incomplete decision table may allow too little behaviour (it forbids too much)!

- This very incomplete decision table:

<table>
<thead>
<tr>
<th>$T$: room ventilation</th>
<th>$r_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$ button pressed?</td>
<td>$\times$</td>
</tr>
<tr>
<td>$\neg on$ ventilation off?</td>
<td>$\times$</td>
</tr>
<tr>
<td>$on$ ventilation on?</td>
<td>$-$</td>
</tr>
<tr>
<td>$go$ start ventilation</td>
<td>$\times$</td>
</tr>
<tr>
<td>$stop$ stop ventilation</td>
<td>$-$</td>
</tr>
</tbody>
</table>

- forbids all actions in case $b \land \neg on \land \neg off$ is satisfied.

- May not be the intention of the customer!
Quality Criteria for DTs: Determinism

**Definition.** [Determinism] A decision table $T$ is called deterministic if and only if the premises of all rules are pairwise disjoint, i.e. if

$$\forall (\varphi_i \rightarrow \alpha_i), (\varphi_j \rightarrow \alpha_j) \in T, i \neq j \Rightarrow \neg (\varphi_i \land \varphi_j).$$

Otherwise, $T$ is called non-deterministic.

**Determinism: Example**

<table>
<thead>
<tr>
<th>$T$: room ventilation</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$ button pressed?</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$-$</td>
</tr>
<tr>
<td>$go$ start ventilation</td>
<td>$\times$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$stop$ stop ventilation</td>
<td>$-$</td>
<td>$\times$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

- is non-deterministic: In a state $\sigma$ with $\sigma \models b$, rules $r_1$ and $r_2$ are both enabled.
- Is non-determinism a bad thing in general?
  - Just the opposite: one of the most powerful modelling tools we have.
  - Read table $T$ as:
    - the button may switch the ventilation on under certain conditions (which I will specify later), and
    - the button may switch the ventilation off under certain conditions (which I will specify later).
  - This is quite some less chaos than full chaos!
  - We can already analyse the specification, e.g., we state that we do not (under any condition) want to see on and off executed together.
Non-determinism: Consequences

- **Good:**
  - A decision table which is intentionally non-deterministic leaves more choices (more freedom) to the developer.

- **Bad:**
  - A non-deterministic decision table leaves more choices to the developer; even the choice to create a non-deterministic final product.
  
  (Input-) Deterministic final products, i.e. “same data in, same data out”, are easier to deal with and are usually desired.
  
- Postponing decisions too long may lead to hasty, bad decisions.

- **Another benefit:**
  - Deterministic decision tables can be implemented with deterministic programming languages.

  For deterministic decision tables, we can easily devise code generation patterns.

Implementing Decision Tables

\[
T: \text{room ventilation} \quad \begin{array}{|c|c|c|}
\hline
b & r_1 & r_2 & \text{else} \\
\hline
\text{button pressed?} & X & X & \text{X} \\
\text{ventilation off?} & \text{X} & \text{X} & \text{X} \\
\text{ventilation on?} & \text{X} & \text{X} & \text{X} \\
\text{start ventilation} & \text{X} & \text{X} & \text{X} \\
\text{stop ventilation} & \text{X} & \text{X} & \text{X} \\
\hline
\end{array}
\]

```c
int b, on, off;
extern void go(); extern void stop();
void (*effect)() = 0;

void dt() {
    read_b_on_off(); // read
    // compute
    if (b && off) effect = go;
    if (b && on) effect = stop;
    execute_effect(); // write
}
```
Domain Modelling for DTs

- Conditions and actions are abstract entities without inherent connection to the real world.
- Yet we want to use decision tables to model/represent requirements on the behaviour of software systems, which are used in the real world.
- When modelling real-world aspects by conditions and actions, we should also model relations between actions/conditions in the real-world (→ domain model (Bjørner, 2006)).

Example:
- if on and off model opposite output values of one and the same sensor for “room ventilation on/off”,
- then $\sigma \models on \land off$ never happens in reality,
- in the abstract setting, $\sigma \models on \land off$ is still possible. $T$ “doesn’t know” that on and off are opposites in the real-world; maybe it should.
- Note: if on and off are outputs of two different, independent sensors, then $\sigma \models on \land off$ is possible in reality (e.g. due to sensor failures).
“Poor Man’s Domain Modelling”

- Add an action cannot happen.

<table>
<thead>
<tr>
<th>$T$: room ventilation</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$ button pressed?</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\star$</td>
<td></td>
</tr>
<tr>
<td>$off$ ventilation off?</td>
<td>$\times$</td>
<td>$-$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>$on$ ventilation on?</td>
<td>$-$</td>
<td>$\times$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>$go$ start ventilation</td>
<td>$\times$</td>
<td>$-$</td>
<td>$-$</td>
<td></td>
</tr>
<tr>
<td>$stop$ stop ventilation</td>
<td>$-$</td>
<td>$\times$</td>
<td>$-$</td>
<td></td>
</tr>
<tr>
<td>$ch$ cannot happen</td>
<td></td>
<td></td>
<td></td>
<td>$\times$</td>
</tr>
</tbody>
</table>

- **Pro:**
  - old definition of completeness applies

- **Con:**
  - all actions are equal, the action ‘ch’ is more equal
  - well-formedness property: no other actions are used with ‘ch’
  - ‘ch’ is in the computation paths of $T$

---

**Conflict Axioms for Domain Modelling**

- A conflict axiom for conditions $C$ is a formula $\varphi_{conf} \in \Phi(C)$.

  **Intuition:** a conflict axiom characterises all those cases, i.e. all those combinations of condition values, which ‘cannot happen’ according to our understanding of the domain.

**Standard semantics wrt. conflict axiom:**

- Let $T$ be a decision table over $C$ and $A$. The set $[[T, \varphi_{conf}]]_{interleave}$, the (standard) interleaving semantics/interpretation of $T$ under conflict axiom $\varphi_{conf}$, consists of the finite or infinite computation paths

  $\pi = \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$, $\sigma_i \in \{\sigma \in \Sigma | \sigma \not\models \varphi_{conf}\}$, $i \in \mathbb{N}_0$,

  where $\forall i \in \mathbb{N}_0 \exists r = \varphi \rightarrow \alpha \in T \bullet \sigma_i \models \varphi \land \alpha_{i+1} = \alpha$.

  **Note:** states satisfying $\varphi_{conf}$ do not occur in $\pi$. 
Vacuity, Completeness, Etc. With Conflict Axiom

Definition. [Vacuity wrt. Conflict Axiom] A rule \( r = \varphi \rightarrow \alpha \in T \) over \( C \) and \( A \) is called vacuous wrt. conflict axiom \( \varphi_{\text{conf}} \in \Phi(C) \) if and only if the premise of \( r \) implies the conflict axiom, i.e. if \( \models (\varphi \implies \varphi_{\text{conf}}) \).

- Intuition: A vacuous rule would only be enabled in states which 'cannot happen'.

Definition. [Completeness wrt. Conflict Axiom] A decision table \( T \) is called complete wrt. conflict axiom \( \varphi_{\text{conf}} \) if and only if the disjunction of the conflict axiom and all rules' premises is a tautology, i.e. if

\[
\models \varphi_{\text{conf}} \lor \bigvee_{\varphi \rightarrow \alpha \in T} \varphi.
\]

- Intuition: A complete decision table cares for all cases which 'may happen'.

- Note: with \( \varphi_{\text{conf}} = \text{false} \), we obtain the previous definitions as a special case. Fits intuition: \( \varphi_{\text{conf}} = \text{false} \) means we don't exclude any states from consideration.

Example: Conflict Axioms

- Let \( \varphi_{\text{conf}} = (\text{on} \land \text{off}) \lor (\neg\text{on} \land \neg\text{off}) \).

"\text{on} models an opposite of \text{off}, neither can both be satisfied nor bot non-satisfied"

- Then

<table>
<thead>
<tr>
<th>( T ): room ventilation</th>
<th>( r_1 )</th>
<th>( r_2 )</th>
<th>( r_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>button pressed? ( b )</td>
<td>( x )</td>
<td>( x )</td>
<td>-</td>
</tr>
<tr>
<td>ventilation off? ( \text{off} )</td>
<td>( x )</td>
<td>-</td>
<td>( x )</td>
</tr>
<tr>
<td>ventilation on? ( \text{on} )</td>
<td>-</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>start ventilation ( \text{go} )</td>
<td>( x )</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>stop ventilation ( \text{stop} )</td>
<td>-</td>
<td>( x )</td>
<td>-</td>
</tr>
</tbody>
</table>

is complete wrt. \( \varphi_{\text{conf}} \).

- Advantage: no conditions 'hidden' in the else-rule.
Conflict Axiom: Consequences

- **Vacuity** wrt. $\varphi_{\text{conf}}$:
  
  Same as with uselessness and general vacuity, **doesn’t hurt** but
  
  May make communication with customer harder!
  
  Implementing vacuous rules is a waste of effort!

- **Incompleteness**:
  
  An incomplete decision table may allow too little behaviour (it forbids too much)!
  
  May **not be the intention of the customer**!


- to stop a plane after touchdown, there are **spoilers** and **thrust-reverse systems**,
- enabling one of those while in the air, can have **fatal consequences**,
- **design decision**: the **software should block** activation in this case,
- **spoilers**: at least one of (i) and (ii) must be true; **thrust-reverse**: only if (i) true.
  
  (i) at least 6.3 tons **weight on each main landing gear strut**,  
  (ii) the **wheels** of the plane must be **turning faster than 133 km/h**.

- **domain model**: “it cannot happen, that (i) and (ii) are not satisfied on ground”.

14 Sep. 1993:

- tower announced crosswind,
- actually tailwind,
- anti-crosswind manoeuvre puts too little weight on landing gear
- wheels didn’t turn fast due to hydroplaning.
In certain domains, we may not want to execute certain actions together.

Let’s declare them to be conflicting and define **consistency** also wrt. conflicting actions.

A **conflict relation** on actions $A$ is a **transitive** and **symmetric** relation $\subseteq (A \times A)$.

---

**Definition.** [Consistency] Let $r = \varphi \rightarrow \alpha \in T$ be a rule.

(i) $r$ is called **consistent** with conflict relation $\subseteq$ if and only if there are no conflicting actions in $\alpha$, i.e. if $\not\exists a_1, a_2 \in \alpha \cdot a_1 \subseteq a_2$.

(ii) $T$ is called **consistent** with $\subseteq$ iff all rules $r \in T$ are consistent with $\subseteq$.
Example: Conflicting Actions

- Let $\xi$ be the transitive, symmetric closure of $\text{stop}\xi\text{go}$.
  
  “actions stop and go are not supposed to be executed at the same time”

<table>
<thead>
<tr>
<th>$T$: room ventilation</th>
<th>$r_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$ button pressed?</td>
<td>$\times$</td>
</tr>
<tr>
<td>$\text{off}$ ventilation off?</td>
<td>$-$</td>
</tr>
<tr>
<td>$\text{on}$ ventilation on?</td>
<td>$\times$</td>
</tr>
<tr>
<td>$\text{go}$ start ventilation</td>
<td>$\times$</td>
</tr>
<tr>
<td>$\text{stop}$ stop ventilation</td>
<td>$\times$</td>
</tr>
</tbody>
</table>

- Rule $r_1$ is inconsistent with $\xi$.

Conflicting Actions: Consequences

- **Consistency:**
  A decision table with **inconsistent** rules may do harm in operation!

  Detecting an inconsistency only late during a project can incur significant cost!

  Inconsistencies (in particular in (multiple) decision tables, created and edited by multiple people, as well as in requirements in general) are **not always as obvious** as in the toy examples given here! (would be too easy...)

  And is more difficult to handle with the **collecting semantics**.
Other Semantics for Decision Tables

A Collecting Semantics for Decision Tables

- Let $T$ be a decision table and $\pi = \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$ a state/event sequence.
- Recall: $\pi$ is a computation path of $T$ (in the interleaving semantics) if
  \[ \forall i \in \mathbb{N}_0 \ \exists r = \varphi \rightarrow \alpha \in T \bullet \sigma_i \models \varphi \land \alpha_{i+1} = \alpha. \]
  That is, at each point in time exactly one rule fires, even if $T$ is non-deterministic.
- $\pi$ is a computation path of $T$ (in the collecting semantics) if and only if
  \[ \forall i \in \mathbb{N}_0 \ \exists r = \varphi \rightarrow \alpha \in T \bullet \sigma_i \models \varphi \land \alpha_{i+1} = \bigcup_{\varphi' \rightarrow \alpha' \in T, \sigma_i \models \varphi'} \alpha'. \]
  That is, all rules which are enabled in $\sigma_i$ “fire” simultaneously, the joint effect is the union of the effects.
- Advantage:
  - separation of concerns, multiple (smaller) tables may contribute to a transition,
  - no non-determinism between rules: all enabled ones “fire”.
- Disadvantages: conflicts much less obvious.
**Consistency in The Collecting Semantics**

**Definition.** [Consistency in the Collecting Semantics] Let $T$ be a decision table. $T$ is called consistent with conflict relation $\triangleright$ in the collecting semantics if and only if there are no conflicting actions in the effect of jointly enabled transitions, i.e. if

$$\triangleright \varphi_1 \rightarrow \alpha_1, \varphi_2 \rightarrow \alpha_2, \sigma \in \Sigma \bullet$$

$$\sigma \models \varphi_1 \land \varphi_2 \land \exists a_1, a_2 \in \alpha_1 \cup \alpha_2 \bullet a_1 \triangleright a_2.$$  

**An Update Semantics for Decision Tables**

- By now, we didn’t talk about the effect of actions from $A$ on states. Actions are uninterpreted.

- Recall the “cash cheque” example:

  Here it makes sense, because the next state seen by the bank clerk may be the result of many database updates by other bank clerks, not only hers/his.

We can also define a semantics with action effects:

- Let $C$ a set of conditions and $A$ a set of actions, $\Sigma = C \rightarrow B$ the standard states.

- Let $\llbracket \cdot \rrbracket_{acteff} : A \times \Sigma \rightarrow \Sigma$ which assigns to each pair of $(a, \sigma)$ of action and state a new state $\sigma'$. $\sigma'$ is called the result of applying $a$ to $\sigma$.

- Example: on $\Sigma = \{b, on, off\}$, we could define

  $$\llbracket go \rrbracket_{acteff} (\sigma) = \{ b \mapsto \sigma(b), on \mapsto 1, off \mapsto 0 \}$$

  $$\llbracket stop \rrbracket_{acteff} (\sigma) = \{ b \mapsto \sigma(b), on \mapsto 0, off \mapsto 1 \}$$

- The interleaving semantics with action effects then requires

  $$\forall i \in \mathbb{N}_0 \exists r = \varphi \rightarrow \alpha \in T \bullet \sigma_i \models \varphi \land \alpha_{i+1} = \alpha \land \sigma_{i+1} = \llbracket \alpha \rrbracket_{acteff} (\sigma).$$
• In addition, we may want to constrain initial states, i.e. give a set $\Sigma_{ini} \subseteq \Sigma$.
• Computation paths of $T$ (over $\Sigma$) are then required to have $\sigma_0 \in \Sigma_{ini}$.

• **Example**: decision table

<table>
<thead>
<tr>
<th>$T$: room ventilation</th>
<th>$r_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>off ventilation off?</td>
<td>$\times$</td>
</tr>
<tr>
<td>on ventilation on?</td>
<td>$-$</td>
</tr>
<tr>
<td>go</td>
<td>$\times$</td>
</tr>
<tr>
<td>stop</td>
<td>$-$</td>
</tr>
</tbody>
</table>

with $\Sigma_{ini} = \{ \{ on \mapsto 0, off \mapsto 1 \} \}$ has only one computation path, namely $\sigma_0 \overset{\text{off}}{\Rightarrow} \sigma_1$ with $\sigma_1 = \{ on \mapsto 0, off \mapsto 1 \}$.

• We can say $T$ terminates.
• This gives rise to another notion of vacuity: $r = (on \land off) \rightarrow \alpha$ is never enabled, because no state satisfying the premise is ever reached (even with conflict axiom $\varphi_{conf} = \text{false}$).

**Distinguishing Controlled and Uncontrolled Conditions**

• For some systems, we can distinguish inputs and outputs.
• **In terms of decision tables**:
  • $C$ is partitioned into **controlled conditions** $C_c$ and **uncontrolled conditions** $C_u$, i.e. $C = C_c \cup C_u$.
  • actions only affect controlled conditions.

• **Example**:
  • $C_c = \{ on, off \}$ (only the software switches the ventilation on or off),
  • $C_u = \{ b \}$ (the button is not controlled by the software, but by the environment, by a user external to the computer system)

• One more quality criterion, **another notion of completeness**:
  We want the specification to be able to deal with all possible sequences of inputs, i.e. we require $[T]|_{C_u} = (\Sigma|_{C_u})^\omega$ (or $(\Sigma|_{C_u})^\ast$).
Example:

<table>
<thead>
<tr>
<th>T: room ventilation</th>
<th>r₁</th>
<th>r₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>button pressed?</td>
<td>×</td>
<td>-</td>
</tr>
<tr>
<td>ventilation off?</td>
<td>×</td>
<td>-</td>
</tr>
<tr>
<td>ventilation on?</td>
<td>-</td>
<td>×</td>
</tr>
<tr>
<td>start ventilation</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>stop ventilation</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

- **is not input sequence complete:**
There is no rule enabled if the button is pressed when the ventilation is off.

- **Note:** it’s not that pressing the button such a state has no effect, but the system stops to work; it “gets stuck” in that state.

  (Because in order to take a transition, we need to have at least one enabled rule.)

**Decision Tables: Discussion**
Decision Tables Summary

- **decision tables** are **one (very simple) example** for a **formal** requirements specification language with
  - **formal syntax** (there are even two: the formula and the table notation)
  - **formal semantics** (also multiple: interleaving, collecting, update)
  - a **formal proof system** — naja, not a dedicated one

- A requirements specification in form of **decision tables** allows us to **formally** reason about properties of what is being developed.
  We can, e.g., **prove** that a decision table is complete. (Automatically?)

- Whether a **decision table** is useful in a particular software development project (of course) **depends** on the project.

- Like many **formal specification language**,
  - a **decision table** may not be the right tool for all problems,
  - it may be tedious to specify all requirements using **decision tables** — don’t do that then.

- One particular **drawback** of **decision tables**:
  - they don’t scale so well in the number of conditions.

Speaking of Formal Methods

“Es ist aussichtslos, den Klienten mit formalen Darstellungen zu kommen; […]”
(“It is futile to approach clients with formal representations”)
(Ludewig and Lichter, 2013)

- ... **of course it is** — vast majority of customers is not trained in formal methods.
Speaking of Formal Methods

“Es ist aussichtslos, den Klienten mit formalen Darstellungen zu kommen; [...]”
("It is futile to approach clients with formal representations")
(Ludewig and Lichter, 2013)

- ...of course it is — vast majority of customers is not trained in formal methods.
- formalisation is (firstly) for developers — analysts have to translate for customers.
- formalisation is the description of the analyst’s understanding, in a most precise form.
- precision: whoever reads it whenever to whomever, the meaning will not change.
- Recommendation: (Course’s Manifesto?)
  - use formal methods for the most important/intricate requirements,
  - use formalisms that you know (really) well,
  - you may use different formalisms for different requirements (if you know what you’re doing!)
  - trying to formalise all requirements is in most cases futile.

References
References


