Today's Menu

Last Seminar: Feature Orientation (Sandeep led)
Project Topic Clarifications

This Seminar: Requirements Reuse (Nash is leading)

Next Seminar: Design & Architecture Reuse (Tanmay leads)

PRS: Beyond “Product Singularity”

→ Most RE techniques focus on individual models
  % Build a model, get it consistent and complete, then validate it%
  % Assumes that RE is a process with a single definite output%
  > The output is a complete, consistent, valid specification of the requirements.
  % This ignores reality!
  % Requirements Engineering isn’t just about obtaining a specification%
  % Requirements are volatile; changes need to be managed continuously%
  % The specification is never complete anyway%
  % There is never just one model:
    > There are multiple versions of models over time%
    > There are multiple variants of models that explore different issues%
    > There are multiple components of models representing different decompositions%
    > Families of models evolve over time (add, delete, merge, restructure the family)
  % RE must address requirements evolution%
  > How do we manage incremental change to requirements models?
  > How can multiple models (specifications) be compared?
  > How will changes to a model affect the properties established for it?
  > How do you capture the rationale for each change?
  > How do we reason about inconsistent and incomplete models?

Why Reusing “Requirements”?

→ Krueger’90,#1 Software Reuse Truism

#1: For a SW Reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation.

% Source code implementation vs. Requirements specification%
  > Cognitive distance is reduced by one level in the SW development life cycle: how ↔ what%
% Effective reuse occurs at the right level of abstraction%
  > E.g., matrix multiplication, sine, substring Subroutines%
  > E.g., stack, list, tree ADTs%
  > E.g., singleton, decorator, iterator, visitor Design Patterns%
  > E.g., pipe & filter, client-server, event-driven Architectural Styles%
  > E.g., PRS, problem frames, FAPs%

Reuse-Focused Requirements Modeling

→ Feature modeling [Kang et al 1990]
→ Orthogonal variability modeling [Pohl et al 2005]
Use Cases

- Domain use case diagram [Moon et al 2005]
- UC with OVM constructs [Halmans & Pohl 2003]

Scenarios

- "Episode": a named subsequence shared among several scenarios [Alspaugh et al 1999]

Goal Models

- Characterizing every goal’s OR-decompositions [Liaskos et al 2006]

Statecharts

- Variability in condition and action [Nejati et al 2007]
The following slides are about SCR, which the PRS paper uses in the case study.
SCR basics

→ Modes and Mode classes
   % A mode class is a finite state machine, with states called system modes
   % Transitions in each mode class are triggered by events
   % Complex systems are described using a number of mode classes operating in parallel

→ System State
   % A (system) state is defined as:
     % the system is in exactly one mode from each mode class
     % and each variable has a unique value

→ Events
   % An event occurs when any system entity changes value
   % An input event occurs when an input variable changes value
   % Single input assumption - only one input event can occur at once
   % Notation: @T(c) means “c changed from false to true”

→ A conditioned event is an event with a predicate
   % @T(c) WHEN d means: “c became true when c was false and d was true”

Source: Adapted from Heitmeyer et al. 1996.

Defining Mode Classes

→ Mode Class Tables
   % Define a (disjoint) set of modes (states) that the software can be in.
   % A complex system will have many different modes classes
   % Each mode class has a mode table showing the events that cause transitions between modes

→ Example:

<table>
<thead>
<tr>
<th>Current Mode</th>
<th>Powered on</th>
<th>Too Cold</th>
<th>Temp OK</th>
<th>Too Hot</th>
<th>New Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>@T</td>
<td>@T</td>
<td>@T</td>
<td>-</td>
<td>Inactive Heat</td>
</tr>
<tr>
<td>@T</td>
<td>-</td>
<td>-</td>
<td>@T</td>
<td>@T</td>
<td>Heat</td>
</tr>
<tr>
<td>@T</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>AC</td>
</tr>
<tr>
<td>@T</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Off</td>
</tr>
<tr>
<td>Ac</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Inactive</td>
</tr>
<tr>
<td>AC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Inactive</td>
</tr>
</tbody>
</table>

Source: Adapted from Heitmeyer et al. 1996.

Defining Controlled Variables

→ Event Tables
   % defines how a controlled variable changes in response to input events
   % Defines a partial function from modes and events to variable values

<table>
<thead>
<tr>
<th>Modes</th>
<th>@C(target)</th>
<th>never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat, AC</td>
<td>@C(target)</td>
<td>never</td>
</tr>
<tr>
<td>Inactive, Off</td>
<td>-</td>
<td>@C(target)</td>
</tr>
<tr>
<td>Ack_tone</td>
<td>@C(target)</td>
<td>-</td>
</tr>
</tbody>
</table>

→ Condition Tables
   % defines the value of a controlled variable under every possible condition
   % Defines a total function from modes and conditions to variable values

<table>
<thead>
<tr>
<th>Modes</th>
<th>@C(target)</th>
<th>@C(target)</th>
<th>never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat</td>
<td>target - temp ≤ 5</td>
<td>target - temp &gt; 5</td>
<td></td>
</tr>
<tr>
<td>AC</td>
<td>temp - target ≤ 5</td>
<td>temp - target &gt; 5</td>
<td></td>
</tr>
<tr>
<td>Inactive, Off</td>
<td>true</td>
<td>never</td>
<td></td>
</tr>
<tr>
<td>Warning light</td>
<td>Off</td>
<td>On</td>
<td></td>
</tr>
</tbody>
</table>

Source: Adapted from Heitmeyer et al. 1996.

Refresher: FSMs and Statecharts

→ on hook
→ off hook
→ Dialtone
→ Busytone
→ Ringtone
→ Connected

Source: Adapted from Heitmeyer et al. 1996.
**SCR Equivalent**

<table>
<thead>
<tr>
<th>Current Mode</th>
<th>offhook</th>
<th>dial</th>
<th>callee offhook</th>
<th>New Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>@T</td>
<td>-</td>
<td>-</td>
<td>Dialtone</td>
</tr>
<tr>
<td>Dialtone</td>
<td>-</td>
<td>@T</td>
<td>F</td>
<td>Ringtone</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>@T</td>
<td>T</td>
<td>Busytone</td>
</tr>
<tr>
<td>Busytone</td>
<td>@F</td>
<td>-</td>
<td>-</td>
<td>Idle</td>
</tr>
<tr>
<td>Ringtone</td>
<td>-</td>
<td>-</td>
<td>@T</td>
<td>Connected</td>
</tr>
<tr>
<td></td>
<td>@F</td>
<td>-</td>
<td>-</td>
<td>Idle</td>
</tr>
<tr>
<td>Connected</td>
<td>-</td>
<td>-</td>
<td>@F</td>
<td>Dialtone</td>
</tr>
<tr>
<td>AC</td>
<td>@F</td>
<td>-</td>
<td>-</td>
<td>Idle</td>
</tr>
</tbody>
</table>

→ Interpretation:

- In Dialtone: @T(offhook) WHEN callee_offhook takes you to Ringing
- In Ringtone: @F(offhook) takes you to Idle
- Etc...

**State Machine Models vs. SCR**

→ All 3 models on previous slides are (approx) equivalent

→ State machine models

- Emphasis is on states & transitions
  - No systematic treatment of events
  - Different event semantics can be applied
  - Graphical notation easy to understand (?)
  - Composition achieved through statechart nesting
  - Hard to represent complex conditions on transitions
  - Hard to represent real-time constraints (e.g. elapsed time)

→ SCR models

- Emphasis is on events
  - Clear event semantics based on changes to environmental variables
  - Single input assumption simplifies modelling
  - Tabular notation easy to understand (?)
  - Composition achieved through parallel mode classes
  - Hard to represent real-time constraints (e.g. elapsed time)