Lecture 16: Modelling “events”

- Focus on states or events?
  - E.g., SCR table-based models
  - Explicit event semantics
- Comparing notations for state transition models
  - FSMs vs. Statecharts vs. SCR
- Checking properties of state transition models
  - Consistency Checking
  - Model Checking, using Temporal Logic
- When to use formal methods

Tabular Specifications: SCR

### Dictionaries:
- Monitored/Controlled Variables
- Types
- Constants

### Tables:
- Mode Transition Tables
- Event Tables
- Condition Tables

Also:
- Assertions
- Scenarios

### SCR Specification

What are we modelling?

- Starting point:
  - States of the environment
  - Events that occur in the application domain (that change the state of the environment)
- Requirements expressed as:
  - Constraints over states and events of the application domain
  - E.g., “When the aircraft is in the air, the pilot should be prevented from accidentally engaging the reverse thrust”
- To get to a specification:
  - For each relevant application domain event, find a corresponding input event
  - For each relevant state, ensure there is a way for the machine to detect it
  - For each required action, find a corresponding output event

SCR basics

- Modes and Mode classes
  - A mode class is a finite state machine, with states called system modes
  - Complex systems described using several mode classes operating in parallel
- System State is defined as:
  - The system is in exactly one mode from each mode class
  - And each variable has a unique value
- Events
  - Single input assumption - only one input event can occur at once
  - An event occurs when any system entity changes value
  - An input event occurs when an input variable changes value
- Notation:
  - $\Diamond c$ expresses that $c$ is true in the environment
  - $\Box c$ expresses that $c$ is always true
  - $c$ is true in the current state
- A conditioned event is an event with a predicate
  - $\Diamond \Box (\langle c \rangle \psi)$ WHEN $d \equiv c \land c \land d$
## 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.
  - A mode table defines a partial function from modes and events to 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>-</td>
<td>-</td>
<td>Inactive</td>
</tr>
<tr>
<td></td>
<td>@T</td>
<td>-</td>
<td>@T</td>
<td>-</td>
<td>Heat</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AC</td>
</tr>
<tr>
<td>Inactive</td>
<td>@F</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Heat</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AC</td>
</tr>
<tr>
<td>Heat</td>
<td>@F</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inactive</td>
</tr>
<tr>
<td>AC</td>
<td>@F</td>
<td>-</td>
<td>@T</td>
<td>-</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inactive</td>
</tr>
</tbody>
</table>

### Example:

- Define how a controlled variable changes in response to input events.
- Defines a partial function from modes and events to variable values.

### Event Tables

- Defines a partial function from modes and events to variable values.

### Condition Tables

- Defines the value of a controlled variable under every possible condition.
- Defines a total function from modes and conditions to variable values.

### Refresher: FSMs and Statecharts

- **Idle**
- **Dialtone**
- **Busytone**
- **Ringtone**
- **Connected**

### SCR Equivalent

- **Current Mode**
- **offhook**
- **dial**
- **callee offhook**
- **New Mode**

### Interpretation:
- In **Dialtone**: @T(offhook) WHEN callee_offhook takes you to Ringtone
- 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)

E.g. Consistency Checks in SCR

→ Syntax
  % did we use the notation correctly?

→ Type Checks
  % do we use each variable correctly?

→ Disjointness
  % is there any overlap between rows of the mode tables?
    → ensures we have a deterministic state machine

→ Coverage
  % does each condition table define a value for all possible conditions?

→ Mode Reachability
  % is there any mode that cannot ever happen?

→ Cycle Detection
  % have we defined any variable in terms of itself?

Formal analysis

→ Consistency analysis and typechecking
  % “Is the formal model well-formed?”
    → [assumes a modeling language where well-formedness is a useful thing to check]

→ Validation:
  % Animation of the model on small examples
  % Formal challenges:
    → “if the model is correct then the following property should hold…”
  % What if questions:
    → reasoning about the consequences of particular requirements;
    → reasoning about the effect of possible changes

→ State exploration
  % E.g. use a model checking to find traces that satisfy some property

→ Verifying design refinement
  % “does the design meet the requirements?”

Model Checking

→ Has revolutionized formal verification:
  % emphasis on partial verification of partial models
    → E.g. as a debugging tool for state machine models
  % fully automated

→ What it does:
  % Mathematically – computes the “satisfies” relation:
    → Given a temporal logic theory, checks whether a given finite state machine is a
      model for that theory
  % Engineering view – checks whether properties hold:
    → Given a model (e.g. a FSM), checks whether it obeys various safety and liveness
      properties

→ How to apply it in RE:
  % The model is an (operational) Specification
    → Check whether particular requirements hold of the spec
  % The model is an abstracted portion of the Requirements
    → Carry out basic validity tests as the model is developed
  % The model is a conjunction of the Requirements and the Domain
    → Formalise assumptions and test whether the model respects them
Model Checking Basics

→ Build a finite state machine model
  % E.g. PROMELA - processes and message channels
  % E.g. SCR - tables for state transitions and control actions
  % E.g. RSML - statecharts + truth tables for action preconditions

→ Express validation property as a logic specification
  % Propositions in first order logic (for invariants)
  % Temporal Logic (for safety & liveness properties)
    > E.g. CTL, LTL, ...

→ Run the model checker:
  % Computes the value of: model |= property

→ Explore counter-examples
  % If the answer is 'no' find out why the property doesn't hold
  % Counter-example is a trace through the model

Example

The solution:

Example

→ Build a finite state machine model

→ Express validation property as a logic specification

→ Run the model checker:

→ Explore counter-examples

→ Sample Properties
  % If you are connected you can hang up:
  % AG(CONNECTED) → EX(~OFFHOOK)
  % If you are connected, hanging up always disconnects you:
  % AG(CONNECTED) → AX(~OFFHOOK → ~CONNECTED))
  % A connection doesn’t start until you pick up the phone:
  % AG(~CONNECTED) → AG(~CONNECTED U OFFHOOK)
  % If you make a call, the phone cannot ring without returning to idle first:
  % AG(RINGTONE ∧ BUSYONE) → A~RINGTONE U IDLE)

→ Temporal Logic

→ LTL (Linear Temporal Logic)

→ CTL (Computational Tree Logic)

→ Complexity Issues

→ The problem:
  % Model checking is exponential in the size of the model and the property
  % There are many propositional variables
  % Current MC engines can explore $10^{100}$ states...
  %...highly optimized data structures (BDDs)

→ The solution:
  % Abstraction:
  > Replace related groups of states with a single state
  > Replace real & integer variables with propositional variables
  % Projection:
  > Slice the model to remove parts unrelated to the property
  > Compositional verification - break large model into smaller pieces
  > (But it's hard to verify that the composition preserves properties)
Formal Methods in RE

→ Why formalize in RE?
  % models of requirements knowledge (so we can reason about them)
  % specifications of requirements (so we can document them precisely)

Why people don’t formalize in RE
  % Formal Methods tend to be lower level than other analysis techniques
  % Formal Methods tend to concentrate on consistent, correct models
  % People get confused about which tools are appropriate:
    > E.g. modeling program behavior vs. modeling the requirements
  % Formal methods require more effort
  > and the payoff is deferred

Selective use of Formal Methods
  % Amount of formality can vary
  % Need not build complete formal models
    > Apply to the most critical pieces
    > Apply where existing analysis techniques are weak
  % Need not formally analyze every system property
    > E.g. check safety properties only
  % Need not apply FM in every phase of development
    > E.g. use for modeling requirements, but don’t formalize the system design
  % Can choose what level of abstraction (amount of detail) to model

FM in practice

→ From Shuttle Study [Crow & DiVito 1996]
  % More errors found in the process of formalizing the requirements than were
  found in the formal analysis
    > Formalization forces you to be precise and explicit, hence reveals problems
  % Formal analysis then finds fewer, but more subtle problems
  % Typical errors found include:
    > inconsistent interfaces
    > incorrect requirements (system does the wrong thing in response to an input)
    > clarity/maintainability problems

<table>
<thead>
<tr>
<th>Issue Severity</th>
<th>With FM</th>
<th>Existing</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Major</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Low Major</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>High Minor</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>Low Minor</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Totals</td>
<td>30</td>
<td>4</td>
</tr>
</tbody>
</table>