Lecture 8, Part 1: Modelling "State"

- What is State?
  - statespace for an object
  - concrete vs. abstract states

- Finite State Machines
  - states and transitions
  - events and actions

- Modularized State machine models: Statecharts
  - superstates and substates
  - Guidelines for drawing statecharts

Getting objects to behave

- All objects have "state"
  - The object either exists or it doesn’t
  - If it exists, then it has a value for each of its attributes
  - Each possible assignment of values to attributes is a "state"
    - (and non-existence is a state, although we normally ignore it)
- E.g. For a stack object

Abstraction

- The state space of most objects is enormous
  - State space size is the product of the range of each attribute
    - E.g. object with five boolean attributes: 2^5 + 1 states
    - E.g. object with five integer attributes: (maxint)^5 + 1 states
    - E.g. object with five real-valued attributes: ...
  - If we ignore computer representation limits, the state space is infinite
- Only part of that state space is "interesting"
  - Some states are not reachable
  - Integer and real values usually only vary within some relevant range
  - We’re usually not interested in the actual values, just certain ranges:
    - E.g. for Age, we may be interested in age<18, 18<=age<=65; and age>65
    - E.g. for Cost, we may only be interested in cost=budget, cost=0, cost>budget,
      and cost>(budget+10%)
Collapsing the state space

The abstraction usually permits more traces
- E.g. this model does not prevent traces with more pops than pushes
- But it still says something useful

Is this model indicative or optative?

What are we modelling?

Application Domain
- D - domain properties
- C - computers
- P - programs

Machine Domain
- R - requirements

The world vs. the machine
StateCharts

Notation:
- States
  - "Interesting" configurations of the values of an object's attributes
  - may include a specification of action to be taken on entry or exit
  - States may be nested
  - States may be "on" or "off" at any given moment
- Transitions
  - Are enabled when the state is "on"; disabled otherwise
  - Every transition has an event that acts as a trigger
  - A transition may also have a condition (or guard)
  - A transition may also cause some action to be taken
  - When a transition is enabled, it can fire if the trigger event occurs and it guard is true
- Events
  -occurrence of stimuli that can trigger an object to change its state
  - determine when transitions can fire

Syntax: event [guard] / action

Events

- occurrence of stimuli that can trigger an object to change its state
- determine when transitions can fire

States can be nested, to make diagrams simpler
- States can be nested, to make diagrams simpler
- A superstate consists of one or more states.
- Superstates make it possible to view a state diagram at different levels of abstraction.

OR superstates
- when the superstate is "on", only one of its substates is "on"

AND superstates
- when the superstate is "on", all of its substates are also "on"
- Usually, the AND substates will be nested further as OR superstates

雇用
- employed
  - probationary
    - after [6 months]
- full

雇用
- employed
  - on payroll
  - assigned to project

States in UML

- A state represents a time period during which
  - A predicate is true
    - e.g. (budget - expenses) > 0,
  - An action is being performed, or an event is awaited:
    - e.g. checking inventory for order items
    - e.g. waiting for arrival of a missing order item
- States can have associated activities:
  - do/activity
    - carries out some activity for as long as the state is "on"
  - entry/action and exit/action
    - carry out the action whenever the state is entered (exited)
  - include/stateDiagramName
    - "calls" another state diagram, allowing state diagrams to be nested

A more detailed example
Events in UML

Events are happenings the system needs to know about:
- Must be relevant to the system (or object) being modelled.
- Must be modelisable as an instantaneous occurrence (from the system’s point of view).
  - E.g., completing an assignment, failing an exam, a system crash.
- Are implemented by message passing in an OO Design.

In UML, there are four types of events:
- **Change events** occur when a condition becomes true.
  - Denoted by the keyword 'when': e.g., \( \text{when} \{ \text{balance < 0} \} \).
- **Call events** occur when an object receives a call for one of its operations to be performed.
- **Signal events** occur when an object receives an explicit (real-time) signal.
- **Elapsed-time events** mark the passage of a designated period of time:
  - e.g., \( \text{after} \{ 10 \text{ seconds} \} \).

Checking your Statecharts

**Consistency Checks**
- All events in a statechart should appear as:
  - Operations of an appropriate class in the class diagram.
- All actions in a statechart should appear as:
  - Operations of an appropriate class in the class diagram.

**Style Guidelines**
- Give each state a unique, meaningful name.
- Only use superstates when the state behaviour is genuinely complex.
- Do not show too much detail on a single statechart.
- Use guard conditions carefully to ensure statechart is unambiguous.
  - Statecharts should be deterministic (unless there is a good reason).

**You probably shouldn’t be using statecharts if:**
- You find that most transitions are fired "when the state completes".
- Many of the trigger events are sent from the object to itself.
- Your states do not correspond to the attribute assignments of the class.

What are we modelling?

**Application Domain**
- D - domain properties
- R - requirements

**Machine Domain**
- C - computers
- P - programs

- **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.
**Four Variable Model:**

- **System:**
  - Monitored/Controlled: true
  - Output: true
  - Input: true
  - Software
- **Controlled Variables:**
  - Waterlevel
  - Temperature
  - Warning light
  - AC power
- **Monitored/Controlled Variables:**
  - Too Hot
  - Too Cold
  - Inactive
- **Constants:**
  - TimeNow
  - AC Failure
  - Heat Failure
  - No Failure
  - Buzzer
  - Warning light
  - Heater
  - Ack_tone

**SCR Specification**

**Definitions:**
- **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 to events to modes.

**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 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.

**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.

**Defining Mode Classes**

- **Event Tables**
  - Defined how a controlled variable changes in response to input events.
  - Defines a partial function from modes and events to variable values.

**Condition Tables**

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

**Modifications**

- **Mode Transition Tables**
  - Used to denote values after the event.
  - We may need to refer to both the old and new value of a variable.

- **Event Tables**
  - Monitored/Controlled: true
  - Output: true
  - Input: true
  - Software

**Examples**

- **Event Tables**
  - Off: @T t - t - Inactive Heat AC
  - Inactive: @F - @T - Off Heat AC
  - Heat: @F - @T - Off Inactive
  - AC: @F - @T - Off Inactive

- **Condition Tables**
  - Heat: target - temp > 5 target - temp > 5
  - AC: temp - target > 5 temp - target > 5
  - Inactive: Off never
  - Warning light: Off On

**Notation:**
- Used primed values to denote values after the event.
- @T(y=1) = y \prime = 1 \ y = 1
- A conditioned event is an event with a predicate:
  - @T(c) WHEN d \ w \ c \ d

**Source:** Adapted from Heitmeyer et al. 1996.
Refresher: FSMs and Statecharts

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)

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></td>
<td>@F</td>
<td>-</td>
<td>-</td>
<td>Idle</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>@F</td>
<td>-</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.

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
- Checking application properties:
  "will the system ever do the following..."

Verifying design refinement
- "Does the design meet the requirements?"
### 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?

### 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

### 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

### Temporal Logic

- **LTL (Linear Temporal Logic)**
  - Expresses properties of infinite traces through a state machine model
  - Adds two temporal operators to propositional logic:
    - $p \rightarrow q$ - $p$ is true eventually (in some future state)
    - $\square p$ - $p$ is true always (now and in the future)
- **CTL (Computational Tree Logic)**
  - Branching-time logic - can quantify over possible futures
  - Each operator has two parts:
    - $EX p$ - $p$ is true in some next states
    - $AX p$ - $p$ is true in all next states
    - $EF p$ - along some path, $p$ is true in some future state
    - $AF p$ - along all paths...
    - $E[p U q]$ - along some path, $p$ holds until $q$ holds:
    - $Ap U qj$ - along all paths...
    - $E[p U q]$ - along some path, $p$ holds in every state:
    - $A[p U q]$ - along all paths...
The solution:
- Abstraction: Replace related groups of states with a single state.
- Projection: Slice the model to remove parts unrelated to the property.
- Compositional verification: Break large model into smaller pieces.

Complexity Issues
- Model Checking is exponential in the size of the model and the property.
- Current MC engines can explore $10^{320}$ states.
  - Using highly optimized data structures (BDDs).
  - And state space reduction techniques.
- That's roughly 400 propositional variables.
- Integer and real variables cause real problems.
- Realistic models are often too large to be model checked.

The problem:
- Model Checking is exponential in the size of the model and the property.
- Current MC engines can explore $10^{320}$ states.
  - Using highly optimized data structures (BDDs).
  - And state space reduction techniques.
- That's roughly 400 propositional variables.
- Integer and real variables cause real problems.
- Realistic models are too large to be model checked.

Formal Methods in RE
- What to formalize in RE?
  - Models of requirements knowledge (so we can reason about them).
  - Specifications of requirements (so we can document them precisely).
- Why formalize in RE?
  - Remove ambiguity and improve precision.
  - Provide a basis for verification that the requirements have been met.
  - Reason about the requirements.
    - Properties of formal requirements models can be checked automatically.
    - Can test for consistency, explore the consequences, etc.
  - Can animate/execute the requirements.
  - Helps with visualization and validation.
- Why people don't formalize in RE?
  - Formal Methods require more effort.
  - By the time your models are consistent, correct models are appropriate:
    - E.g., modeling program behavior vs. modeling the requirements.
  - Formal methods advocates get too attached to one tool.

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.
  - 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>
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