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
What does the model mean?

- **Finite State Machines**
  - There are a finite number of states (all attributes have finite ranges)
    - E.g. imagine a stack with max length = 3
  - E.g. imagine a stack with max length = 3
  - E.g. new();Push();Push();Top();Pop();Push();
  - E.g. no trace can start with a Pop()
  - E.g. no trace may have more Pops than Pushes
  - E.g. no trace may have more than 3 Pushes without a Pop in between

- The model specifies a set of traces
  - E.g. new();Push();Push();Top();Pop();Push();
  - E.g. new();Push();Pop();Push();Pop();
  - E.g. no trace can start with a Pop()
  - E.g. no trace may have more Pops than Pushes
  - E.g. no trace may have more than 3 Pushes without a Pop in between

- The model excludes some behaviours
  - E.g. no trace can start with a Pop()
  - E.g. no trace may have more Pops than Pushes
  - E.g. no trace may have more than 3 Pushes without a Pop in between

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: $(\text{maxint})^5+1$ states
    - E.g. object with five real-valued attributes: $\ldots$?
    - 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

What are we modelling?

- Observed states of an application domain entity?
  - E.g. a phone can be idle, ringing, connected, ...
  - Model shows the states an entity can be in, and how events can change its state
  - This is an indicative model

- Required behaviour of an application domain entity?
  - E.g. a telephone switch shall connect the phones only when the callee accepts the call
  - Model distinguishes between traces that are desired and those that are not
  - This is an optative model

- Specified behaviour of a machine domain entity?
  - E.g. when the user presses the 'connect' button the incoming call shall be connected
  - Model specifies how the machine should respond to input events
  - This is an optative model, in which all events are shared phenomena
Is this model indicative or optative?

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.
  - Syntax:  \texttt{event [guard] / action}

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

---

Superstates

- 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** (concurrent substates)
- When the superstate is "on", all of its states are also "on"
- Usually, the AND substates will be nested further as OR superstates
A more detailed example

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
Events in UML

Events are happenings the system needs to know about

- Must be relevant to the system (or object) being modelled
- Must be modellable 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. when(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. after[10 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 and

- 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
Lecture 8, Part 2: 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

What are we modelling?

Application Domain                Machine Domain

- 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
Tabular Specifications: SCR

**Four Variable Model:**

- **System**
  - Input devices
  - Input data items
  - Software
  - Output data items
  - Output devices
  - Controlled variables
  - Environment

**Dictionaries:**

- Monitored/Controlled Variables
- Types
- Constants

**Tables:**

- Mode Transition Tables
- Event Tables
- Condition Tables

**SCR Specification**

**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
  - Notation:
    - We may need to refer to both the old and new value of a variable:
    - Used primed values to denote values after the event
    - $\ominus T(c) = \neg c \land c'$
    - $\ominus F(c) = \neg c \land \neg c'$
  - A conditioned event is an event with a predicate
    - $\ominus T(c) \ WHEN \ d = \neg c \land c' \land d$

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.
  - 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>-</td>
<td>t</td>
<td>-</td>
<td>Inactive</td>
</tr>
<tr>
<td></td>
<td>@T</td>
<td>t</td>
<td>-</td>
<td>-</td>
<td>Heat</td>
</tr>
<tr>
<td></td>
<td>@T</td>
<td>-</td>
<td>-</td>
<td>t</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>@T</td>
<td>-</td>
<td>-</td>
<td>Heat</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>@T</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>- @T</td>
<td>-</td>
<td>Inactive</td>
</tr>
<tr>
<td>AC</td>
<td>@F</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>@T</td>
<td>-</td>
<td>Inactive</td>
</tr>
</tbody>
</table>

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.
  - Example:

<table>
<thead>
<tr>
<th>Modes</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat, AC</td>
<td>@C(target)</td>
</tr>
<tr>
<td>Inactive, Off</td>
<td>never</td>
</tr>
<tr>
<td>Ack_tone =</td>
<td>Beep</td>
</tr>
<tr>
<td></td>
<td>Clang</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.
  - Example:

<table>
<thead>
<tr>
<th>Modes</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat</td>
<td>target - temp ≥ 5</td>
</tr>
<tr>
<td></td>
<td>target - temp &gt; 5</td>
</tr>
<tr>
<td>AC</td>
<td>temp - target ≥ 5</td>
</tr>
<tr>
<td></td>
<td>temp - target &gt; 5</td>
</tr>
<tr>
<td>Inactive, Off</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>never</td>
</tr>
<tr>
<td>Warning light =</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>On</td>
</tr>
</tbody>
</table>
Refresher: FSMs and Statecharts

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>-</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)

---

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

- 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

Temporal Logic

- LTL (Linear Temporal Logic)
  - Expresses properties of infinite traces through a state machine model
  - adds two temporal operators to propositional logic:
    - ♣p - p is true eventually (in some future state)
    - □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;
    - A[p U q] - along all paths...
    - EG p - along some path, p holds in every state;
    - AG p - along all paths...
### Example

- **offhook**
- **busytone**
- **idle**
- **dialed**
- **connected**
- **ringtone**
- **on hook**
- **dialtone**

#### Sample Properties

- If you are connected you can hang up:
  \[\text{AG}(\text{CONNECTED} \Rightarrow \text{EX}(\neg \text{OFFHOOK}))\]
- If you are connected, hanging up always disconnects you:
  \[\text{AG}(\text{CONNECTED} \Rightarrow \text{AX}(\neg \text{OFFHOOK} \rightarrow \neg \text{CONNECTED}))\]
- A connection doesn’t start until you pick up the phone:
  \[\text{AG}(\neg \text{CONNECTED} \Rightarrow \text{A}[\neg \text{CONNECTED} \cup \text{OFFHOOK}])\]
- If you make a call, the phone cannot ring without returning to idle first:
  \[\text{AG}((\text{RINGTONE} \lor \text{BUSYTONE}) \Rightarrow \text{A}[(\neg \text{RINGING} \cup \text{IDLE})])\]

### Complexity Issues

#### The problem:

- Model Checking is exponential in the size of the model and the property
- Current MC engines can explore \(10^{120}\) 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 solution:

- **Abstraction**:
  - Replace related groups of states with a single superstate
  - 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?
- Remove ambiguity and improve precision
- Provides a basis for verification that the requirements have been met
- Can 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
- Will have to formalize eventually anyway
  - RE is all about bridging from the informal world to a formal machine domain

Why people don’t formalize in RE
- Formal Methods tend to be lower level than other analysis techniques
  - They force you to include too much detail
- Formal Methods tend to concentrate on consistent, correct models
  - ...but most of the time your models are inconsistent, incorrect, incomplete...
- People get confused about which tools are appropriate:
  - E.g., modeling program behaviour vs. modeling the requirements
  - formal methods advocates get too attached to one tool!
- Formal methods require more effort
  - ...and the payoff is deferred

What to formalize in RE?
- models of requirements knowledge (so we can reason about them)
- specifications of requirements (so we can document them precisely)

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>
Using Formal Methods

- 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