Testing State Machines

(refers to Binder chapter 7)

Premises

- OO software is well suited to state-based testing:
  - A state machine is a system whose output is determined by both current and past input. In contrast, the output of a combinational system is determined by current input only.
    - » eFSMs have state variables
  - A system has state-based behavior when identical inputs are not always accepted and, when accepted, may produce different outputs.
  - Definitions p.178
  - Modus operandi p.179: no concurrency and potentially ignored inputs.
- State-based testing applies to several scopes:
  - Methods
  - Classes
  - Subsystems: Binder's 'mode machine' (e.g., for containers)
- Different notations in Figure 7.3. (p.182)
- Binder proposes the FREE method and the N+ test design strategy

Properties of FSMs

- Formal models require an explicitly specified transition for ever possible event/state pair. Otherwise we have an incomplete specification.
  - Most state models are incomplete specifications!!
- Two (set of) states could be equivalent:
  - Occurrence of an equivalence typically indicates a flaw.
- Non-reachable states could exist:
  - Dead states: no out transitions
  - Dead loop: no state outside the loop can be reached and the loop prevents reaching the final state
  - Magic state: only out transitions (implies redundancy with initial state)
- Guards are used to avoid non-determinism:
  - State “Loaded” does not work in Figure 7.4 (p.184)
  - Figure 7.5 corrects this problem (p.185)

Mealy vs Moore

- Mealy:
  - States are passive
  - See Fig 7.6 (incomplete specification) (p.186)
    - » e.g., p1_Start() is ignored in most states (i.e., deemed illegal)
- Moore:
  - Actions are in states
  - See fig 7.7 (p.188): more complex … because every action/state pair must be represented with a state. If several actions are used in different combinations, each combination requires a unique state…
- Interesting to see the list of procedures of the 2-player game on p.187:
  - Bridges between FSMs and code: the state model defines valid sequences of procedure calls.
- Mealy is preferred but they are mathematically equivalent.
- Layout problems exist for “large” state machines…

Table Formats

- State-to-state: fig 7.8 (p.190)
- Event-to-state: fig 7.9 (p.191)
  - Paired event/state format…
- Expanded state-to-state: fig 7.10 (.192)
  - Separate resulting actions (top) from resulting state (bottom

- These formats:
  - are better than others wrt testability:
    - » Discussion of other possible implementations (pp.202-204)
      - Revisits Go4's 'state' pattern
  - exemplify our inability to model concurrency using simple state machines:
- Instead, Product machines represent all combinations of concurrent states… (p.195-7)
    - » Hierarchical statecharts are product machines in disguise.
      - Compare Fig 7.11 to Fig 7.12 (or 7.13 for a gradual viewpoint)

More on Concurrency

- Usual semantics of statecharts:
  - Substates are in an XOR decomposition relation
  - AND decomposition is represented using orthogonal states:
    - » See figure 7.14 (p.199)
    - » Can be re-expressed using a basic state machine (fig 7.15 p.200) but the number of states in the product machine is bounded by the product of the number of states in each component machine.
- Harel: statecharts = state diagrams + depth + orthogonality + broadcast
  - Terminological explanations p.200
    - » Broadcast: the output of a process (i.e., hierarchical state) can be sent to and consumed by another process
    - » Concurrent objects and orthogonal states are quite different!
- There are semantic glitches:
  - e.g.: instantaneous transitions open the door to an event possibly negating itself…
- And UML has its own semantics:
  - Table 7.1 p.202 is important!!

An Introduction to FREE

- FREE: Flattened Regular Expression
  - Restricted to "pure" Mealy machines (i.e. no hybrid)
  - Flattened to represent behavior of *inherited* features
    - » ROSE-RT uses flattened FSMs wrt hierarchical states
  - Every state machine as an equivalent regular expression
  - A UML model that follows the FREE conventions will be testable
  - Object state at method activation and deactivation is the focus of the FREE method: States result from the computation done by a method.
- A usable OO state model must answer several questions:
  - p.204: back to the basic semantics
- A testable model has specific criteria:
  - p.205: second bullet is difficult to know a priori

States?

- Nebulous definitions for a state: footnote p.205
- For state-based test suites, we need to decide whether a test case has (or has not) produced the expected state.
  - An executable definition is required for effective automation…
  - The states of a class are subsets of the set of all possible combinations of instance variable values. These subsets are limited to the acceptable results of computations done by methods and are grouped according to the way in which they determine acceptable message sequences.
    - » A state is therefore a relationship among instance variable values that can be coded as a Boolean expression in the OOPL of the implementation under test…
  - Example: figure 7.16 (p.207)
    - » Key point: if we only care about overdrawn, inactive and neither as states determining behavior than fig 7.17 (p.208) shows us how to partition our representation.
      - Good luck doing this in more than 3 dimensions… ☹
  - State invariants may be useful!!
    - » Tied to class invariants and method post-conditions
      - See top paragraph p.209: a state invariant defines a subset of what the class invariant allows…

More Definitions

- **Instance variable domain:**
  - set of discrete values (state) that may be represented by an instance variables.
- **Combination value set:**
  - Set of all possible combinations of instance variable values
- **Visibility:**
  - Abstract: object viewed as a black-box
  - Concrete: object viewed from an implementation standpoint
- **Scope:**
  - Bounded: only abstract values of state variables are considered
  - Recursive: all levels of abstraction are flattened until only primitive data types remain…
- **Granularity:**
  - Aggregate: states may consist of an arbitrary number of members of the combinational value set
  - Primitive: each member of the combinational value set is viewed as a set.
- **Leads to a classification of states: (p.210)**
  - Can you understand table 7.2! It all about how you view the data members of an instance…

Some Specifics of FREE

- **The UML hybrid is error-prone and effort-intensive. It is thus prohibited in FREE (p.212)**
  - Really technical footnote p.212: why Moore machines are bad because we would have to check how much of the code has been executed.
- **An event is:**
  - Either a message sent to the class under test (c-u-t)
  - A response received from a server of the c-u-t
  - An interrupt or similar external control action that must be accepted by the c-u-t
- **A guard has no side effects and can refer to self but must account for scope (client -> external methods only)**
- **An action is:**
  - Either a message sent to a server object
  - The response provided by an object of the c-u-t to its client
  - The structure of complex output actions (with loops and conditions) can be modeled with regular expressions
- **Constructors and the notion of initial state(s) are difficult to reconcile:**
  - Alpha state: declaration BEFORE construction
  - Alpha transition: a constructor
  - Omega state: object destroyed or out of scope

Subtyping

- Subclasses are to be subtypes
  - The state space of the superclass MUST NOT BE mutilated!
    - » Additional subclass states are orthogonal to states of the superclass
    - » Bullets p.215: can get a bit obscure…
- Contravariance (footnote p.215) has an impact on assertion writing
  - Meyer has a book on how complicated this is…
  - Summarized in figure 7.18 (p.216)
- Flattening inheritance may introduce orthogonal states (see next slide), which, in turn, need to be translated into a single product machine (of all possible combinations)…
  - See figure 7.24 (p.223): flattened 7.19 (p.217)…
  - Flattening may not always be necessary (p.222):
    - » Only test subclass with inherited features
    - » Do not have a model of superclass

Inheritance and State Machines

For subtyping, the subclass may use:

- Orthogonal composition (fig 7.19): additional “features”…
  - » Do you understand the orthogonal states and their relation to constructors? Wht Y/X(int) and not Y/X()??
    - Remember that an instance of Z has the data members of X and Y…
- Concatenation (fig 7.20): e.g. mixin
- State partitioning and substate addition (fig 7.21)
  - » C is decomposed and J is added.
  - » Notice how protocol with D is still obeyed but methods 4 and 6 have been overridden
- Transition retargeting (fig 7.22)
- Transition splitting (fig 7.23)

- The handling of constructors is not obvious…
  - Why not show us code examples?
  - Can we reasonably expect designers to work out by hand such flattened statecharts??

The Response Matrix

- A sneak path is the bug that allows an illegal transition (from an illegal event that should have been ignored) or eludes a guard.
- Concurrency compounds the problem: an event may need to be ignored by one concurrent state and not by another…
- FREE insists on the use of a response matrix (on a flattened statechart)
  - Consider fig 7.25 (p.225)
  - Figure 7.26 (p.226) gives the corresponding response matrix which uses the action codes of table 7.3
    - » Notice how we are back to an *exhaustive* set of Boolean combinations for guards…
  - p.228: robustness (ie defensive code) vs cooperative design…
    - » Binder claims defensive classes localize state control

Control Faults

- General list p.229 + exhaustive table 7.4 p.230
- Examples figs 7.27-34 (pp.231-235)
- Must go beyond local scope to catch bugs related to inheritance (p.236):
  - The bulleted list indicates how tricky things can be…
- Use of checklists at design time is still an effective strategy!
  - Table 7.5 (p.238) Structure: item 1.6 is surprising
  - Table 7.6 (State names): items 1.3, 1.7.2, 1.7.3 are useful!
  - Table 7.7 (Guards): item 3.3 is important!
  - ALL of table 7.8 is crucial for the use of inheritance with statecharts!
  - Table 7.9 defines what a complete specification really entails (by including robustness considerations)!

The N+ Strategy

not on the exam

- This strategy
  - Requires flattened state model
  - Exercises all implicit transitions (for sneak paths)
  - Depends on ability to report on resultant state
- Recipe p.242
- Example: 3 player volley…
  - Models on pp.244-5 response matrix on p. 246
  - Relies on round-trip paths: sequences that begin and end with the same state AND simple paths from initial to final state.
    - » 1 iteration of a loop
    - » Treatment of guards p.247
      - Back to combinational strategies all true combinations + one false…
    - » Transform your FSM graph into a tree (BFT or DFT) (fig 7.38)!!!!!!!
    - » Generate tests (table 7.10 pp.251-2): from root to leaves
  - Sneak path analysis (only for incomplete specs)
    - » From response matrix (recipe pp.253-4)
    - » One test case for each non-checked, non-excluded transition cell
    - » See table 7.11: relies on ability to put in a certain state…
  - Path sensitization (values to exercise guards) is undecidable…
    - » Must be solved heuristically…

State-Based Strategies (1)

- Several strategies have been proposed for state-based testing:
  - <u>Once</u> you understand the individual strategies:
    - » See the power hierarchy fig 7.39 and comparison fig 7.40
    - » Comparing N+ and W methods wrt size of test suite (table 7.12)
- The strategies are:
  - Piecewise:
    - » Test at least once each state, each event and each action…
    - » Quite useless…
  - All transitions:
    - » Every specified transition is exercised at least once
    - » Necessarily exercises all states, events and actions
    - » No particular sequencing of transitions is considered
    - » Catches incorrect and missing event/action pairs
    - » Unless we have a complete specification, sneak paths (ie extra transitions) are not revealed
    - » Cannot show incorrect resulting state

State-Based Strategies (2)

- All n-transition sequences:
  - Test sequences of n events
- All round-trip paths: (p.261)
  - sequences that begin and end with the same state (including self-transitions)
  - "Any sequence that goes beyond a round trip must be part of a sequence that belongs to another round-trip. Thus, a test suite that achieves all-round-trip path coverage will reveal all incorrect or missing event/actions pairs."
  - Corresponds to a n-switch cover, that is, to all transition sequences of length k <= n
  - Cannot guarantee incorrect or invalid states are all revealed given such a state could occur as a result of a n+1 sequence of events…
- M-Length signature:
  - For truly opaque entities-under-test: A state signature consists of a sequence of output actions that are unique for a particular starting state. It is extracted from the specs…
  - A signature of a certain length can uniquely identify a state under certain conditions…
  - It's not worth developing such signatures according to Binder given we can build state inspectors and thus avoid this strategy completely!

Indispensable Tools

- **Checking the resultant state:**
  - Reporter methods
    - Typically have state-set and state-get methods
    - Some have tried a defeasible checksum approach
    - Simplest state-get evaluates the state invariant and returns a Boolean indicating that an object is or is not in that state.
  - Test repetition for corrupt states:
    - Only if you do not have reporters
    - Repeat a test sequence and compare the sequences of actions performed…
  - State revealing signatures (of output actions) (p.258)
    - As previously mentioned: very hard and costly to obtain…
    - Abandoned in favor of B-I-Ts in VLSI
  - Assertions (pre- and post- conditions as well as invariants, all attached to states) could be used for B-I-T.
    - Assertions are NOT limited to state-based testing and are discussed at length later in the course.