State-Based Testing

Generating test cases for complex behaviour

Motivation
- We are interested in testing the behaviour of object-oriented software systems
- Behaviour: Interactions (message passing) between objects
- State machines can model such behaviour at any scope (class, cluster, system)
- If our test model is a state machine, we can generate test cases for the modeled behaviour

What is a state machine?
- A system whose output is determined by both current and past input
- Previous inputs are represented by the state
- State-based behaviour: Identical inputs are not always accepted. When accepted, they may produce different outputs

Building blocks of a state machine
- **State**: An abstraction that summarizes past inputs, and determines behaviour on subsequent inputs
- **Transition**: An allowable two-state sequence. Caused by an event
- **Event**: An input or a time interval
- **Action**: The output that follows an event

State machine behaviour
1. Begin in the **initial state**
2. Wait for an event
3. An event comes in
   - If not accepted in the current state, ignore
   - If accepted, a transition fires, output is produced (if any), the **resultant state** of the transition becomes the current state
4. Repeat from step 2 unless the current state is the **final state**

State machine properties
- How events are generated is not part of the model
- Transitions fire one at a time
- The machine can be in only one state at a time
- The current state cannot change except by a defined transition
- States, events, transitions, actions cannot be added during execution
State machine properties

- Algorithms for output creation are not part of the model
- The firing of a transition does not consume any amount of time

The challenge: How to model the behaviour of a given system using a state machine?

Guarded transitions

- The previous model is ambiguous, e.g. there are two possible reactions to push and pop in the Loaded state
- Guards can be added to transitions
- A guard is a predicate associated with the event
- A guarded transition cannot fire unless the guarded predicate evaluates to true

Guarded transitions

Limitations of the basic model

- Not specific for object-oriented systems
- Limited scalability
  - Even with the best tools available, diagrams with 20 states or more are unreadable
- Concurrency cannot be modeled
  - Different processes can be modeled with different state machines, but the interactions between them cannot

Scalability example
Statecharts

Statechart advantages
- Easier to read
- Suited for object oriented systems (UML uses statecharts)
- Hierarchical structure helps with state explosion
- They can be used to model concurrent processes as well...

Concurrent statechart

State model
- Must support automatic test generation
- The following criteria must be met
  - Complete and accurate reflection of the implementation to be tested
  - Allows for abstraction of detail
  - Preserves detail that is essential for revealing faults
  - Represents all events and actions
  - Defines state so that the checking of resultant state can be automated

What is a state?
- We need an executable definition that can be evaluated automatically
- An object with two boolean fields has 4 possible states?
  - This would lead to trillions of states for typical classes
- Instead, state is: A set of variable value combinations that share some property of interest
- Can be coded as a boolean expression
An example

- Consider the following class

```java
class Account {
    AccountNumber number;
    Money balance;
    Date lastUpdate;
}
```

- A primitive view of the state space would yield too many states...

Trillions of states

A valid state can be expressed with a state invariant, a boolean expression that can be checked

- A state invariant defines a subset of the values allowed by the class invariant

State invariants

- Ensure a or b in Eiffel normally defines two states

Three abstract states

- A transition is a unique combination of
  - Two state invariants (one for the accepting, one for the resultant state - may be the same)
  - An associated event
  - An optional guard expression
  - An optional action(s)

Transitions

Events

1. A message sent to the class under test
2. A response received from a supplier of the class under test
3. An interrupt or similar external control action that must be accepted

Events
Guards

- A predicate expression associated with an event
- Depending on the context of the model, guards may or may not reference private features
- Evaluation of the guard must not have side effects

Actions

1. A message sent to a supplier object
2. The response provided by an object of the class under test to its client

Alpha and Omega states

- The initial stage of an object is the state right after it is constructed
- However, a class may have multiple constructors that leave the object at different states
- To avoid modeling problems we define that an object is in the $\alpha$ state just before construction
- Similarly with $\omega$ and destruction (not necessary to model $\omega$ for languages that have garbage collection)

Expanding the statechart

- Statecharts are great for communication, reducing clutter etc.
- They might hide subtle bugs (e.g. entering a substate rather than a superstate)
- We need to expand them to full transition diagrams for testing purposes
- An automatable process

Unspecified Event/State Pairs

- State machine models will not include all events for all states
- Implicit transitions may be illegal, ignored, or a specification omission
- Accepted illegal events lead to bugs called sneak paths
- For testing purposes, we cannot ignore implicit behaviour

Example statechart
Possible responses to illegal events

<table>
<thead>
<tr>
<th>Response Code</th>
<th>Name</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Accept</td>
<td>Perform the explicitly specified transition</td>
</tr>
<tr>
<td>1</td>
<td>Queue</td>
<td>Place the illegal event in a queue for subsequent evaluation and ignore</td>
</tr>
<tr>
<td>2</td>
<td>Ignore</td>
<td>No action or state change is to be produced, no error is returned, no exception raised</td>
</tr>
<tr>
<td>3</td>
<td>Flag</td>
<td>Return a non-zero error code</td>
</tr>
<tr>
<td>4</td>
<td>Reject</td>
<td>Raise an IllegalEventException</td>
</tr>
<tr>
<td>5</td>
<td>Mute</td>
<td>Disable the source of the event and ignore</td>
</tr>
<tr>
<td>6</td>
<td>Abend</td>
<td>Invoke abnormal termination services (e.g., core dump) and halt the process</td>
</tr>
</tbody>
</table>

Interesting aside

- Illegal events can be coded either using defensive programming or contracts
- We saw many benefits of contracts in 3311
- One benefit of defensive programming with regard to testing is that testing can concentrate on the supplier side (clients are allowed to send any message)

State model validation

- A state model must be complete, consistent, and correct before it is used to generate test cases
- Five validation checklists apply
  - Structure checklist
  - State name checklist
  - Guarded transition checklist
  - Flattened machine checklist
  - Robustness checklist

Structure checklist

- There is an initial state with only outbound transitions
- There is a final state with only inbound transitions (if not, explicit reason is needed)
- No equivalent states
- Every state is reachable from the initial state
- The final state is reachable from all states
- Every defined event and every defined action appears in at least one transition

Structure checklist

- Except for the initial and final states, every state has at least one incoming and one outgoing transition
- The events accepted in a particular state are unique or differentiated by mutually exclusive guards
- Complete specification: For every state, every event is accepted or rejected (either explicitly or implicitly)
State name checklist

- Poor names are often indications of incomplete or incorrect design
- Names must be meaningful in the context of the application
- If a state is not necessary, leave it out
- "Wait states" are often superfluous
- State names should be passive
- Adjectives are best, past participles are OK

Guarded transition checklist

- The entire range of truth values for a particular event is covered
- Each guard is mutually exclusive of all other guards
- Guard variables are visible
- Guards with three or more variables are modeled with a decision table
- The evaluation of a guard does not cause side effects

Robustness checklist

- There is an explicit spec for an error-handling or exception-handling mechanism for implicitly rejected events
- Illegal events do not corrupt the machine (preserve the last good state, reset to a valid state, or self-destruct safely)
- Actions have no side effects on the resultant state
- Explicit exception, error logging, and recovery mechanisms are specified for contract violations

Fault model for state machines

- Two general types of error
  - Control faults: An incorrect sequence of events is accepted, or an incorrect sequence of outputs is produced
  - Incorrect composite behaviour: Errors associated with inheritance

Control faults

- Missing transition
- Incorrect transition
- Missing action
- Incorrect action
- Sneak path
- Corrupt state
- Illegal message failure
- Trap door

Missing transition
The N+ Test Strategy

- Test strategies for state machines may include: Exhaustive, All Transitions etc.
- The N+ Test strategy:
  - Develop a state-based model of the system
  - Validate the model using the checklists
  - Expand the statechart
  - Develop the response matrix
  - Generate the round-trip path test cases
  - Generate the sneak path test cases
  - Sensitize the transitions in each test case

The 3-player game example

- We will use an extension of the 2-player game as an example
- There is also a third player that may win any of the volleys

Round-Trip Path Tree

- Exercise all transitions and loops on every possible alpha-omega path at least once
- Root: Initial stage
- Edge for each transition
- Stop if the resultant state is already in the tree or is a final state
- Guards: One transition for each variant that evaluates to True
- Counter Guards: Transition annotated with *
Transition tree for the 3-player game

Transition tree tests explicit behaviour
We need to test each state's illegal events
A test case for each non-checked, non-excluded transition cell in the response matrix
Confirm that the actual response matches the specified response

Test Case Input | Test Event | Expected Result
---|---|---
Player 1 Started | p1_start | IllegalEventException
Player 2 Started | p2_start | IllegalEventException
Player 3 Started | p3_start | IllegalEventException
Player 1 Won | p1_Won | Player 1 Won
Player 2 Won | p2_Won | Player 2 Won
Player 3 Won | p3_Won | Player 3 Won
Abandoned | | Abandoned