Model Based Testing

Connecting Specifications and Testing
A working definition

- Model-based testing is

“The automatic generation of efficient test procedures/vectors using models of system requirements and specified functionality.”

www.goldpractices.com/practices/mbt/index.php

- There are also benefits of model creation and analysis beyond that of automated test generation, e.g. validation of requirements
- Mostly for integration and acceptance testing
Why a formal model?

• Informal specification documents enable engineers to get vague understanding of system functionality

• Reliance on such implicit, mental, informal models renders testing process that is
  - Unstructured
  - Hardly reproducible
  - Unmotivated in its details

• Informal models cannot support automated test generation and validation
Cost-benefit analysis

- Model creation costs time/money, but:
- Systems get more complex, release schedules shorter
- Automated model-based test generation now possible
- Testing is 50-70% of total cost of product release, clear need to cut that cost factor
- Models can be reused, can correct requirements, can inform design activities

→ Model-based testing often cost-effective but requires certain skills within organization
Possible workflow

1. *Build the model*; e.g. finite-state machine abstraction of system’s event structure
2. *Generate expected inputs*; e.g. trace of events for finite-state machine
3. *Generate expected output*; e.g. target state
4. *Compare* actual *output* with expected one, e.g. was target state reached?
5. *Decide on further actions*; e.g. modify model, generate more tests, estimate reliability
Model building during development

- Requirements engineer, designer, tester, or developer forms mental representation of system’s functionality
- Describes/expands mental model in easily understandable formalism
- Uses formalism and choice of model that facilitate frequent, automated, and effortless test generation
Model creation: other needs

• E.g. *maintenance*: often requires *automated extraction of information from system artifacts*, e.g. from documentation, source code, data files etc.

• Many useful kinds of information: *call graphs*, file dependences, *frequent usage patterns*, event interactions, etc.

• *Example application*: extract event interactions from black-box legacy system, use that model to determine causal structure of events
Kinds of behavioral models, all have tool support

- **Decision tables**: tables showing sets of conditions and actions that result from conditions being true
- **Finite-state machines (FSM)**: finite number of states and transitions (possibly labeled with actions) between them
- **Markov chains**: like finite-state machines but transitions guided by probability distribution
- **State charts**: UML diagram, shows states that system can assume, shows circumstances that cause state change
Example of a state chart
Example FSM model
Example of Markov chain

Diagram showing states and transitions:
- Offline, ready
- Offline, network down
- Connection terminated
- Connection spoofed
- Connected securely

Probabilities and transition rates between states.
Choice of modeling method

- E.g. *use finite-state machines to model state-rich system* such as telephony
- E.g. *use state charts for system with few states*, or hierarchical structure, transitions caused by user input and external conditions
- E.g. *use Markov chains when statistical analysis*, failure data, or reliability assessments are desired
Heuristics for building a model

1. List all inputs

2. For each input: *list situations in which input can be applied*; ditto for situations in which it *cannot be applied*

3. For each input: *list situations in which input causes different behaviors* or outputs, depending on application context of input
Recall FSM model
Details of example FSM model

- FSM is *model of simple phone system*
- Model is of *phone that can call out*
- *Nodes are states of phone*, e.g. OnHook
- *Edges are actions user can take*, i.e. system input, e.g. HangUp
- Test cases specify
  - *sequence of inputs*
  - *states system should reach* after each action
  - and *value of outputs* of system
Generating test cases

OnHook <PickUP> DialTone <Dial/PartyBusy> Busy
<HangUp> OnHook <PickUp> DialTone
<Dial/PartyReady> Ringing … // Exercise: extend sequence to cover all transitions
OnHook <PickUP> DialTone … sequence (from previous slide) has 15 inputs, achieves action coverage: every action possible at each state “executed” at least once; easiest test coverage criterion for FSM model
Action coverage

- Generated *action-coverage sequence not unique*, each such sequence stresses software differently but with same coverage criterion
- Said sequence consists of four test cases, i.e. sequences beginning at *OnHook*
- If system outputs only its abstract state, can *use FSM as effective test oracle*
Switch coverage

• **Switch coverage**: for each state, each pair of actions leading (into, out) of that state is in test sequence

• Switch coverage: *more rigorous than action coverage*

• Example: at DialTone we need to consider $2 \times 3 = 6$ such pairs, e.g. the pair
  
  <$\text{PartyHangsUp}$> DialTone <$\text{Dial/PartyReady}$>

• 26 ($> 15$) inputs needed for switch coverage here
From models to tests & back

- *Models deliberately abstract*: simplification enables comprehension and communication of functionality or requirements.
- *Models generate test cases guided by* coverage criteria, e.g. action coverage, or other *test purposes*, e.g. “Requirement A2”.
- *Generated test cases have to be concrete enough* to be executable: *test scripts/drivers*.
- Executable *test results too concrete to map directly back to models*.

→ *Automation needs to enable move from abstract to concrete and vice versa*.
Test scripts

• Aka test drivers, *run automatically without human interaction*

• *Provide general mechanisms for supporting other test automation methods*

• E.g. capture/playback and test generation approaches

• Test scripts *developable in standard application languages* VB, C, Java, C#, Tcl, …

→ Model-based testing *needs to bridge the gap between abstract models and concrete test scripts*
Common test script pattern

- **Initialize** the SUT
- **Iterate**, for each test case:
  - initialize target (optional)
  - Initialize output to value other than expected (if possible)
  - **Set inputs**
  - **Run SUT**
  - Capture output and state of results so that later on a test report can be created
Capture/playback approach

• Captures sequences of manual operations (e.g. in GUI) in test script written by test engineer
• Has shortcomings, e.g.
  - needs to recognize GUI objects when layout has changed
  - Changing system functionality forces manual recapture of playback sequence
  - Manual recording of today’s website interaction too complex to handle
Model-based testing: benefits

- **Comprehensive tests**: models determine logical paths, locations of program boundaries, identify reachability problems
- **Improved requirements**: testable requirement has to be complete, consistent, unambiguous; testing may expose “feature interaction” requirement defects
- **Defect discovery**: studies suggest mode-based testing results in early defect detection, sufficient for Return On Investment
Some Additional Resources

http://www.geocities.com/model_based_testing/